面向21世纪高等院校计算机系列规划教材
COMPUTER COURSES FOR UNDERGRADUATE EDUCATION

Java 语言程序设计

汤一平 等 编著

科学出版社

北 京

内 容 简 介

本书基于 Java 语言介绍面向对象的程序设计，全书共 12 章，介绍了计算机系统的硬件与软件、Java 程序入门、Java 编程、面向对象的编程技术基础、Applets 和 Graphics、高级的 Java 编程功能、面向对象的高级编程技术、图形化用户界面的编程技术、异常和输入/输出流、Java 的服务器端组件技术和 Java 语言的应用实例——HelkCFG。书中有大量的应用实例，通过本书的学习，读者可独立开发一些 Java 小程序。

本书可以作为计算机、通信等信息类专业本科生的教材，也可以作为广大教学、科研和工程技术人员的参考书。

图书在版编目(CIP)数据

Java 语言程序设计/汤一平等编著. —北京:科学出版社,2006.1
(面向 21 世纪高等院校计算机系列规划教材)
ISBN 7-03-016727-9

Ⅰ.J… Ⅱ.汤… Ⅲ.JAVA 语言-程序设计-高等学校-教材
Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2005)第 157900 号

责任编辑:吕建忠 李 伟/责任校对:刘彦妮
责任印制:吕春珉/封面设计:飞天创意

科学出版社 出版
北京东黄城根北街 16 号
邮政编码:100717
http://www.sciencep.com

印刷
科学出版社发行 各地新华书店经销
*
2006 年 1 月第 一 版 开本:787×1092 1/16
2006 年 1 月第一次印刷 印张:20
印数:1—3 000 字数:453 000

定价:26.00 元
(如有印装质量问题,我社负责调换)
销售部电话 010-62136131 编辑部电话 010-62138978-8001(HI09)

前　言

1991 年，为了开发一种面向家用电器并能够在家用电子产品上进行交互式操作的软件产品，Sun 公司组织了一批优秀的工程师，成立了一个名为 Green 的计算机语言项目开发小组，开发了用于网络的精巧而安全的软件——Oak，它就是 Java 语言的前身。WWW 浏览器的出现使得 Internet 的表现力及魅力陡增，Oak 语言被定位于 WWW 浏览器的应用上。1995 年 1 月，Oak 升级到新的版本并改名为 Java。1995 年春，Sun 公司公布了 Java 的完整技术规范，立即得到包括 Netscape 公司在内的各 WWW 厂商的广泛支持。

由于 Java 语言具有与环境无关、跨平台等特点，这对整个计算机产业产生了深远的影响，对传统的计算模型提出了新的挑战，很快就引起了一场软件革命。

目前，Java 提供 3 个方面的应用：

- J2SE：用于编写桌面工作站应用程序。其最新版本 J2SE 1.5 为企业及客户端应用程序开发提供了更高的性能和更好的 Web 部署。
- J2ME：是致力于消费产品和嵌入式设备的最佳解决方案，特别是在移动通信设备、移动计算设备、小型家电消费产品上得到了广泛的应用。
- J2EE：是前沿的 Java 技术平台，为服务器的计算提供了所有范围的企业级功能。此平台的设计使它能够为创建企业级 n 层Java应用程序提供集成 Java 应用程序环境。

从 Java 所提供的 3 个方面的应用来看，Java 已经渗透到我们生活的方方面面，小到我们所携带的手机，大到航空航天装备。正像 Java 当时作出的最重要的承诺一样，Java 成为了一种“万能胶”，将用户同信息连接到一起，无论这些信息来自 Web 服务器、数据库、信息供应商，还是能够想像到的其他任何一种信息源。由于 Java 拥有坚强的工程技术力量这个后盾，它获得了世界上 IT 厂商的广泛支持。其内建的安全及保密特性可同时满足软件工程师、程序员和最终用户在这方面的需求。而且，Java 语言已经内置了对许多高级编程任务，如网络编程、数据库连接、3D 编程以及多线程等的支持。

目前市场上介绍 Java 的书籍非常多，引进的国外教材和参考书也很多，但是从教与学的角度来审视这些书籍总感到有些不足，笔者感到，在我国高等院校关于计算机科学与技术教育的计算机语言类教材方面还是很有潜力可挖的，这就是编写本书的出发点和目的。

通过本书的课堂授课、上机实践和作业指导，读者能达到以下目标：

1）掌握面向对象程序设计的基本概念、原理和特征。

2）掌握基于 Java 的对象程序设计思想和技术。

3）掌握 Java 语言及工具（主要是 JDK）的使用方法，并能自主开发简单的程序。

4）把在本课程之前学过的关于计算机科学方面的知识通过 Java 编程技术贯通

起来。

一般来说，学习 Java 语言的重点和难点在于纯对象式设计。本书的主要内容是基于 Java 语言进行面向对象的设计，目的是使读者掌握面向对象程序设计的基本概念和原理，并基于 Java 语言学习面向对象程序设计的 3 个重要特征，即封装性、继承性和多态性，最终目标是使读者能把 Java 作为进行面向对象程序设计的一种强有力开发工具。本书的特点之一是既注重理论，也强调软件开发实践；特点之二是便于教与学，其中的每个章节基本上是以一个学时为单位来组织的。

通过本书的学习，读者将了解应用程序，学会创建用于处理终端用户简单输入的基本用户界面，从文件和数据库中读取数据及向文件和数据库写入数据，通过网络发送和接收数据信息。本书并不提供包罗万象的 Java 语言内容，而只提供一些重要的基础性入门指示，以方便读者掌握 Java 平台上可用的一般编程功能。

通过本书的学习，读者基本能开发一些 Java 小程序，以满足 IT 企业的需求。

本书具体编写分工如下，第 1、2、10、11 章由浙江工业大学信息学院的汤一平编写，第 3、6、9 章由浙江林学院的莫路锋编写，第 5、8 章由中国计量学院的唐文彬编写，第 4、7 章由浙江万里学院的陈智罡编写。全书由汤一平统一策划与设计，他还与浙江林学院莫路锋做了最后的校对、完善书稿。

Java 编程语言是一种内容广泛、发展迅速的计算机语言，本书力图用 Java 来介绍计算机科学，这是一种新的教学尝试，以便读者能将有关计算机科学方面的知识通过 Java 编程技术贯通起来。由于笔者的学识和水平有限，本书难免存在不足之处，望读者不吝赐教，以利再版修订。

目　　录

第 1 章　计算机系统的硬件与软件 …… 1

1.1　计算机系统 …… 2

1.1.1　硬件与软件 …… 2

1.1.2　硬件的组成部分 …… 2

1.1.3　存储器 …… 3

1.1.4　辅助存储器 …… 3

1.1.5　输入/输出设备 …… 4

1.1.6　软件 …… 4

1.1.7　程序的种类 …… 5

1.1.8　操作系统 …… 5

1.1.9　网络 …… 6

1.2　模拟信号与二进制信号 …… 7

1.2.1　二进制 …… 7

1.2.2　计算机使用二进制的原因 …… 8

1.2.3　模拟信号 …… 8

1.2.4　二进制信号 …… 9

1.3　计算机存储器 …… 11

1.3.1　存储器的特性 …… 11

1.3.2　信息的存储形式 …… 12

1.3.3　信息的复制 …… 12

1.3.4　字节 …… 12

1.3.5　主存储器 …… 13

1.3.6　硬盘 …… 14

1.3.7　文件 …… 15

1.3.8　文件与操作系统 …… 15

1.3.9　文件的类型 …… 16

1.4　处理器 …… 16

1.4.1　处理器的电子操作 …… 16

1.4.2　机器指令 …… 16

1.4.3　不同的处理器 …… 18

1.4.4　高级编程语言 …… 18

1.4.5　源程序 …… 19

1.4.6　程序的编译 …… 20

1.4.7　可移植性 …… 20

1.4.8　解释程序 …… 21

1.4.9 虚拟机 …… 21
1.4.10 运行速度 …… 22
第 2 章 Java 程序入门 …… 24
2.1 Java 简介 …… 25
2.1.1 安装 Java …… 25
2.1.2 商用 Java 工具软件 …… 25
2.1.3 Java 程序示例 …… 26
2.1.4 字节代码 …… 26
2.1.5 Java 虚拟机 …… 27
2.1.6 Applet …… 27
2.1.7 Java 源程序的创建 …… 28
2.1.8 命令提示符窗口 …… 30
2.1.9 用记事本进行编辑 …… 30
2.1.10 键入源程序 …… 31
2.1.11 保存源文件 …… 31
2.1.12 文件命名 …… 32
2.1.13 Java 程序的运行 …… 32
2.2 Java 小程序 …… 34
2.2.1 示例源程序 …… 35
2.2.2 语法错误 …… 35
2.2.3 修改语法错误 …… 36
2.2.4 编辑、编译、运行三部曲 …… 37
2.2.5 程序漏洞 …… 37
2.2.6 稍长一点的示例程序 …… 37
2.2.7 注释 …… 38
2.2.8 括号 …… 40
2.3 运行示例程序 …… 40
2.4 Java 语言的特性 …… 46
2.4.1 Java 语言的简单性特性 …… 47
2.4.2 Java 语言的面向对象特性 …… 47
2.4.3 Java 语言的分布式计算特性 …… 47
2.4.4 Java 语言的健壮性特性 …… 47
2.4.5 Java 语言的结构中立特性 …… 47
2.4.6 Java 语言的安全性特性 …… 47
2.4.7 Java 语言的可移植特性 …… 48
2.4.8 Java 语言的解释特性 …… 48
2.4.9 Java 语言的多线程功能特性 …… 48
2.4.10 Java 语言的动态功能特性 …… 48
2.4.11 Java 语言与 C 和 C++ 语言的区别 …… 48

第 3 章　Java 编程 …… 51
3.1　基本数据类型 …… 52
3.1.1　数据类型 …… 52
3.1.2　整数类型 …… 52
3.1.3　浮点类型 …… 52
3.1.4　字符类型 …… 53
3.1.5　布尔类型 …… 53
3.1.6　数据类型的封装 …… 53
3.2　变量与常量 …… 54
3.2.1　变量 …… 54
3.2.2　变量的声明 …… 54
3.2.3　变量的命名 …… 54
3.2.4　赋值语句 …… 55
3.2.5　常量 …… 55
3.3　表达式和算术操作符 …… 55
3.3.1　表达式 …… 55
3.3.2　算术运算符 …… 56
3.3.3　数值运算 …… 57
3.4　布尔表达式 …… 59
3.4.1　布尔表达式 …… 59
3.4.2　逻辑操作 …… 59
3.5　简单的 if 语句 …… 60
3.5.1　two-way 判定 …… 60
3.5.2　if . . else 语句 …… 63
3.5.3　单个块 if 语句 …… 63
3.5.4　多个分支选择 …… 64
3.5.5　if 语句的多样性 …… 66
3.5.6　if 语句的嵌套 …… 66
3.6　while 循环和 do 循环 …… 67
3.6.1　while 初涉 …… 67
3.6.2　while 的工作过程 …… 67
3.6.3　while 语句的语法 …… 68
3.6.4　while 语句的语义 …… 69
3.6.5　循环控制变量 …… 69
3.6.6　do. . . while 语句 …… 70
3.6.7　3 件要注意的事情 …… 71
3.7　for 循环 …… 72
3.7.1　循环的 3 部分 …… 72
3.7.2　for 语句 …… 72

3.7.3 for 语句中的循环计数 …… 73
3.7.4 等效的 for 和 while 循环 …… 73
3.7.5 循环控制变量的作用域 …… 74
3.8 输入/输出 …… 75
3.8.1 输入/输出包 …… 75
3.8.2 输入/输出流 …… 75
3.8.3 输入/输出异常 …… 76
3.8.4 数字输入/输出 …… 77
3.8.5 字符输入/输出 …… 78
第 4 章 面向对象的编程技术基础 …… 81
4.1 在 Java 中定义类与对象 …… 82
4.1.1 定义类 …… 82
4.1.2 认识构造函数 …… 85
4.2 类的封装与继承 …… 87
4.2.1 封装 …… 87
4.2.2 类的继承 …… 90
4.3 多态与静态 …… 94
4.3.1 类的多态 …… 94
4.3.2 静态成员的定义与使用 …… 98
4.4 面向对象的基本概念 …… 100
4.4.1 类与对象 …… 100
4.4.2 成员 …… 101
4.4.3 继承 …… 101
4.4.4 多态 …… 102
4.5 重载、屏蔽与覆盖 …… 102
4.5.1 重载 …… 102
4.5.2 屏蔽 …… 104
4.5.3 覆盖 …… 105
第 5 章 Applet 和 Graphics …… 110
5.1 简单的 Applet …… 111
5.1.1 编辑 Applet …… 111
5.1.2 编译 Applet …… 111
5.1.3 运行 Applet …… 112
5.2 Applet 类的层次 …… 113
5.3 Applet 的生命周期 …… 113
5.3.1 Applet 的生命周期举例 …… 113
5.3.2 Applet 的方法 …… 114
5.3.3 Applet 的生命周期和对应的方法 …… 115
5.4 Applet 标记和 HTML …… 116

5.4.1 Applet 标记的语法 …… 116
5.4.2 Applet 标记描述 …… 116
5.4.3 HTML …… 117
5.5 appletviewer …… 119
5.6 绘制图形 …… 119
5.6.1 简单的图形绘制 …… 119
5.6.2 着色 …… 120
5.6.3 绘制直线 …… 120
5.6.4 绘制矩形 …… 121
5.6.5 绘制圆形和椭圆 …… 121
5.7 Circle 类 …… 122
5.8 利用图形方法画图 …… 124
5.8.1 图片的规划 …… 124
5.8.2 计算坐标 …… 125
5.8.3 完整的 Applet …… 126
第 6 章 高级的 Java 编程功能 …… 129
6.1 递增、递减和其他操作符 …… 130
6.1.1 递增/递减操作符 …… 130
6.1.2 赋值操作符 …… 131
6.1.3 调和级数例子 …… 131
6.2 短逻辑运算符 …… 133
6.2.1 位逻辑运算符 …… 133
6.2.2 左移位运算符 …… 135
6.2.3 右移位运算符 …… 136
6.2.4 位运算符赋值 …… 137
6.3 再论 for 语句 …… 138
6.3.1 在 for 循环中声明循环控制变量 …… 139
6.3.2 逗号的使用方法 …… 139
6.3.3 for 循环的一些变化 …… 140
6.3.4 循环嵌套 …… 141
6.4 数组 …… 142
6.4.1 一维数组 …… 142
6.4.2 二维数组和多维数组 …… 144
6.4.3 另一种数组声明语法 …… 148
6.4.4 将数组作为参数 …… 148
6.5 对象的数组 …… 149
6.5.1 对象引用数组 …… 149
6.5.2 字符串引用数组 …… 150
6.5.3 一个例子 …… 150

6.6 线性查找 …… 151
6.6.1 构造函数 …… 156
6.6.2 线性查找 …… 156
6.6.3 完整的查找 …… 157
6.7 向量类 Vector …… 159
6.7.1 向量类 Vector 的引入 …… 159
6.7.2 Vector 类的基础 …… 160
6.7.3 程序举例 …… 163
第 7 章 面向对象的高级编程技术 …… 166
7.1 包的使用 …… 167
7.1.1 package 语句 …… 167
7.1.2 import 语句 …… 168
7.1.3 classpath 的使用 …… 169
7.2 访问控制 …… 170
7.2.1 类成员的访问控制 …… 170
7.2.2 final 限定词 …… 171
7.3 抽象类与接口 …… 171
7.3.1 抽象类 …… 171
7.3.2 接口 …… 173
7.4 对象之间的关系 …… 174
7.4.1 对象间的比较运算 …… 174
7.4.2 对象转为字符 …… 176
7.5 内部类 …… 176
7.5.1 类中定义的内部类 …… 176
7.5.2 内部类被外部类引用的方法 …… 177
7.5.3 匿名内部类 …… 178
7.6 StringBuffer 类与 StringTokenizer 类的使用 …… 179
7.6.1 StringBuffer 类的使用 …… 179
7.6.2 StringTokenizer 类的使用 …… 180
第 8 章 图形化用户界面的编程技术 …… 183
8.1 AWT 及其组件 …… 184
8.1.1 java.awt 包 …… 184
8.1.2 组件 …… 185
8.1.3 容器 …… 186
8.2 创建简单的图形用户界面 …… 187
8.2.1 框架 …… 187
8.2.2 面板 …… 188
8.3 布局管理器 …… 188
8.3.1 FlowLayout 布局管理器 …… 189

8.3.2 BorderLayout 布局管理器 …… 190
8.3.3 GridLayout 布局管理器 …… 192
8.3.4 其他布局管理器 …… 193
8.3.5 容器的嵌套 …… 193
8.4 AWT 事件处理模型 …… 194
8.4.1 事件处理原理 …… 194
8.4.2 事件类 …… 196
8.4.3 事件监听器 …… 197
8.4.4 事件适配器 …… 200
8.5 Swing 简介 …… 202
8.5.1 简介 …… 202
8.5.2 Swing 的类层次结构 …… 204
8.5.3 Swing 包 …… 205
8.5.4 MVC 体系结构 …… 206
8.5.5 可存取性支持 …… 206
8.5.6 支持键盘操作 …… 207
8.5.7 设置边框 …… 207
8.5.8 使用图标 …… 207
8.6 Swing 组件和容器简介 …… 207
8.6.1 Swing 组件 …… 207
8.6.2 JComponent 类 …… 209
8.7 Swing 布局管理器 …… 209
8.8 Swing 程序简介 …… 211
8.8.1 简单的 Swing 源程序 …… 211
8.8.2 使用 Swing 的基本规则 …… 212
8.9 Swing 常用容器组件 …… 212
8.9.1 框架 …… 212
8.9.2 JApplet …… 213
8.9.3 JPanel …… 213
8.9.4 根面板 …… 213
8.9.5 分层面板 …… 214
8.9.6 标签 …… 214
8.9.7 按钮 …… 214
8.9.8 文本输入 …… 215
8.9.9 复选框 …… 219
8.9.10 单选按钮 …… 219
8.9.11 列表框 …… 220
8.5.12 组合框 …… 221
8.9.13 滑块 …… 222

8.9.14 菜单 …… 223
8.9.15 对话框 …… 225
第 9 章 异常和输入/输出流 …… 227
9.1 异常处理 …… 228
9.1.1 异常的基本概念 …… 228
9.1.2 异常类型 …… 228
9.1.3 未被捕获的异常 …… 229
9.1.4 使用 try 和 catch 语句 …… 230
9.1.5 使用多重 catch 语句 …… 231
9.1.6 嵌套 try 语句 …… 233
9.1.7 引发异常 …… 235
9.1.8 throws 语句 …… 236
9.1.9 finally 语句 …… 237
9.1.10 Java 的内置异常 …… 239
9.1.11 创建自己的异常子类 …… 240
9.1.12 使用异常 …… 241
9.2 输入/输出流 …… 241
9.2.1 输入/输出基础 …… 241
9.2.2 流的概念 …… 242
9.2.3 读取控制台输入 …… 244
9.2.4 向控制台写输出 …… 247
9.2.5 PrintWriter 类 …… 248
9.3 文件的读写 …… 249
9.4 文件类 …… 252
9.4.1 目录 …… 254
9.4.2 使用 FilenameFilter …… 255
9.4.3 listFiles( )方法 …… 256
9.4.4 创建目录 …… 256
第 10 章 Java 的服务器端组件技术 …… 258
10.1 EJB 技术介绍 …… 259
10.1.1 EJB 技术 …… 259
10.1.2 EJB 规范的发展历程 …… 260
10.1.3 EJB 的类别 …… 260
10.1.4 EJB 的体系结构 …… 261
10.1.5 EJB 中的角色 …… 262
10.1.6 EJB 与 DCOM/COM + 的比较 …… 263
10.1.7 EJB 与 CORBA 的比较 …… 265
10.1.8 结论 …… 266
10.2 一个 EJB 例子 …… 266

10.2.1　安装 EJB 服务器 …… 267
10.2.2　声明 EJB 远程接口 …… 267
10.2.3　声明主接口 …… 268
10.2.4　编写 EJB 类 …… 268
10.2.5　创建 ejb-jar 文件 …… 270
10.2.6　部署 DemoBean …… 271
10.2.7　编写 EJB 客户机 …… 273
10.2.8　编译并运行客户机程序 …… 275
第 11 章　Java 语言的应用实例——HelkCFG …… 276
11.1　有关系统方面的一些预备知识 …… 277
11.1.1　学而致用 …… 277
11.1.2　编译原理的相关知识 …… 277
11.1.3　数据结构 …… 278
11.1.4　网站的相关知识 …… 279
11.2　HelkCFG 的总体构架 …… 279
11.2.1　HelkCFG 简介 …… 279
11.2.2　HelkCFG 的基本功能 …… 280
11.2.3　网页的链接结构 …… 280
11.2.4　系统的层次结构 …… 281
11.3　系统的设计与实现 …… 282
11.4　编译原理中的词法、语法和语义分析 …… 285
11.4.1　综述 …… 286
11.4.2　词法分析 …… 287
11.4.3　语法分析 …… 290
11.4.4　语义分析 …… 296
11.4.5　包的组织 …… 300
11.5　关于 HelkCFG …… 301
11.5.1　外观界面 …… 301
11.5.2　HelkCFG 涉及的相关知识 …… 301
11.6　使用 HelkCFG …… 303
11.6.1　页面的编排 …… 303
11.6.2　操作范例 …… 303

第 4 章　面向对象的编程技术基础

本章要点

面向对象的基本概念
如何设计与使用 Java 对象

本章学习目标

理解类和对象的基本概念
掌握构造函数的使用方法
理解类的封装、继承与多态
掌握静态成员的使用
理解重载、覆盖与屏蔽
理解面向对象的基本概念

4.1 在 Java 中定义类与对象

4.1.1 定义类

在介绍如何定义类前先看两个例子，认识一下类的基本结构。

下面我们来设计一个名字叫 Person 的类，在 Person 这个类中我们定义 3 个成员，一个是 age 属性，用来存储人的年龄，另外有两个方法，一个是吃东西的方法 eat，另一个是睡觉的方法 sleep。类的定义如下：

例 4.1

```
// Eartch.java
class Person
{
    public int age;
    public void eat ( )
    {
      System.out.println ("Eating…");
    }
    public void sleep ( )
    {
      System.out.println ("Sleeping…");
    }
}
public class Earth
{
    public static void main ( string argv [])
    {
        Person person1 = new Person ( );
        Person person2;
        person2 = new Person ( );
        person1.age = 18;
        System.out.println ("person1:"+ person1.age);
        System.out.println ("person2:"+ person2.age);
        person1.eat ( );
        person2.sleep ( );
    }
}
```


上述代码定义了一个名字叫 Person 的类。第一个 public 这个关键字是一个限定词，限定词的作用是用来限定这个类可不可以让其他的类使用，限定词的具体使用说明将在后面章节中详细说明。前面说过一个 Java 程序文件中，可以同时有好几个类，但是

只有一个类能加上 public 的限定词，而且要求程序的文件名称和限定为 public 的类名称一样。class 关键字说明这是一个类。该代码显示结果如图 4.1 所示。

图 4.1　Person 类的结果显示

该类定义了一个属性 age，属性在 Java 里面其实就是变量，所以可以按照前面学过的变量格式去声明。使用一个变量之前，一定要事先声明，如果没有事先“打招呼”，计算机是找不到要用的变量的，因为它不知道有这个变量。一般我们都把属性的定义放在类定义的最前面。属性 age 前面有 public 限定词，说明其他所有的类都可以访问它。

接下来我们看到该类里定义了两个方法。方法前有个 void 关键词，表示当执行这个方法后，将没有结果返回。但有些时候我们需要方法执行后返回一个结果，然后对这个结果进行处理，这时可在方法前写上返回结果的数据类型就好了。方法名称后面有一个小括号，表明它是一个方法。方法执行的语句放在大括号里。我们定义的这两个方法都很简单，就是在屏幕上输出“Eating…”和“Sleeping…”的字样。

现在我们已经完成了一个类的设计，类可以说只是一个模板，是抽象的，要想拥有一个活生生的具有上述定义的属性和方法的 Person，必须创建具体的对象。就像你到冷饮店想买一个蛋卷冰激凌，售货员会按照蛋卷冰激凌类生产出一个可以吃的蛋卷冰激凌给你。下面来看看如何产生和使用这个类的对象。

要产生一个新的对象，需要使用 new 关键字，例如：

```
Person person1 = new Person ( );
```

赋值号左边表示定义了一个类型是 Person，名字为 person1 的变量。赋值号右边表示通过 new 关键字创建一个 Person 类的实例对象。通过中间的赋值号将这个刚创建好的对象的地址赋给了 person1 的变量，从此就可以通过 person1 来引用该对象。按照 Java 的说法，变量 person1 就是对象的引用，它只是保存对象的地址，就这么简单。至于对象本身，是保存在内存中一个称为“堆”的地方里。就像你把李柔同学的电话号码存在了手机的电话簿里，而且起了一个名字叫“李柔”，以后想联系李柔时，只要在电话簿里找到“李柔”这项，拨叫就可以了。因此在手机电话簿里与“李柔”对应的位置存的是李柔的电话号码，而没有把李柔这个人放在里面。

当一个对象被创建时，会自动对其中各种类型的成员变量进行初始化赋值，如表 4.1 所示。

表 4.1 成员变量类型及初始值

成员变量类型	初始值
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	‘\u0000’(表示为空)
boolean	false
all reference type	null

reference type 指的是引用类型，即类类型。除了基本数据类型之外的变量类型都是引用类型。

创建对象之后，我们可以使用“对象名.对象成员”的格式来访问对象的成员(包括属性和方法)，对象名和对象成员名称之间的“.”，可以把它理解成“的”。前面我们定义的 Person 类虽然完整，但是还不能运行，要让其运行，可以在类中增加一个 main 方法，那么程序就会从 main 方法开始执行程序。所以现在我们又要写一个类，然后在这个类中使用 Person 类，这个新的类就叫 Earth 类吧，因为每个人降生后都属于地球中的一员。

上述代码中，在 Earth.main 方法中创建了两个 Person 类的对象 person1 和 person2，创建这两个对象的格式略有不同，但效果是一模一样的。创建 person1 对象时用了一条语句，定义对象的引用和生成对象一气呵成。而创建 person2 对象时用了两条语句，第一条语句先定义了一个对象的引用，到了第二条语句才真正地生成了 Person 类的对象。有了这两个对象，我们就可以使用它们了。现在 person1 和 person2 是两个完全独立的对象，类中定义的成员变量在每个对象中也是独立的，不会被对象共享，例如改变了 person1.age 的值不会影响 person2.age 的值。接着我们将 person1.age 赋值为 18，然后通过两条输出语句，进行了输出。由于没有给 person2.age 赋值，所以输出的是默认值 0。最后调用了 person1 的 eat 方法和 person2 的 sleep 方法。

从上列我们也就可以概括出类定义的格式及对象定义如下:

```
[<限定词>] class <类名称> [extends <类名称>] [implements <接口名称>]
{
[<代码>]
}

类名称 对象名;
对象名 = new 类名称 ([参数1, 参数2, …]);
```

思考一下，为什么只需要对 Earth.java 这个文件进行编译操作就可以了？

4.1.2　认识构造函数

下面通过例 4.2 来了解构造函数。

例 4.2

```
//Earth.java
class Person
{
  public int age;
  public Person ( )
{
age = 18;
}
  public Person (int a)
{
age = a;
}
  public void eat ( )
  {
        System.out.println ("Eating…");
  }
public void sleep ( )
   {
    System.out.println ("Sleeping…");
  }
}
public class Earth
{
    public static void main ( String argv [])
    {
Person person1 = new Person ( );
Person person2 = new Person (20);
System.out.println ("person1:"+ person1.age);
System.out.println ("person2:"+ person2.age);
person1.age = 25;
System.out.println ("person1:"+ person1.age);
System.out.println ("person2:"+ person2.age);
person1.eat ( );
person2.sleep ( );
        }
    }
```


编译后运行的结果如图 4.2 所示。

图 4.2　例 4.2 的运行结果

要注意的是，当执行 new Person（）的时候，调用的是类 Person 中的 Person（）这个构造函数，而当执行 new Person（20）时，调用的是类 Person 中的 Person（int a）这个构造函数。有些读者可能会问，构造函数名称都一样，编译器怎么知道要调用哪一个构造函数来产生对象呢？虽然上面两个构造函数的名称一样，限定词一样，但仔细端详一下，会发觉参数不一样。一个构造函数没有任何参数，而另一个构造函数传递一个 int 类型的参数。编译器就是靠不同的参数形式来辨别该使用哪一个构造函数。就像我有两个同事都叫张一（相当于两个构造函数），其中一个有小孩（一个构造函数有参数），而另一个没有结婚当然没有小孩啦（另一个构造函数没有参数），如果我在办公室里说："张一，你小孩病好了么？"（调用构造函数），如果此时两个张一都在，我相信大家和两个张一都知道我在和谁说话呢。

我们把构造函数的上述特性称为构造函数的重载，也就是说，在一个类中有多个构造函数使用相同的名称，但是参数类型与个数却各不相同。一定要记住什么是相同，什么应该不同。由于构造函数是一种特殊的方法，对于方法来说也有重载这个概念，关于重载的一般概念，我们会在稍后详细介绍。

到此为止，我们清楚了构造函数的由来，下面再总结一下：

在 Java 的每一个类里都至少有一个构造函数。如果没有人为的在一个类里定义构造函数，系统会自动为这个类产生一个默认的构造函数，这个默认的构造函数没有参数。

如果人为的在一个类里定义了构造函数，系统将不再自动为这个类产生一个默认的构造函数。

构造函数是在用 new 关键字创建一个对象时，由编译器自动调用的，不能直接像使用一般方法那样去调用构造函数。

在例 4.2 中，我们看到生产对象时要通过 new Person（）这条语句，它的功能是产生一个具体的 Person 类对象，其中 Person（）可以看成是一个方法，因为在一个名称后面跟一个小括号，通常是一个方法的调用。于是心细的读者可能产生疑问了：Person（）这个方法在整个程序里都找不到呀，而且这个方法的名称居然和 Person 类的名称一样，这不会是一个巧合吧，Person（）与 Person 类沾亲带故，一定会有一些"皇室血统"。

那么现在可以讲清楚了，这个特殊方法叫构造函数（constructor），顾名思义，它就是构造一个对象的方法。由于类是模板，生产对象必须按照模板进行构造才可以，构造函数就是做这件事的。如果我们自己没有编写构造函数，Java 编译时会为我们自动产

生。构造函数在程序设计中非常有用，它可以为类的成员变量进行初始化工作，当一个类的实例对象被生产出来时，这个类的构造函数就会被自动调用，所以可以在构造函数中加入我们希望对象产生后具有那些特性的代码。就像每个婴儿出生后都要洗个澡，那么可以把“洗澡”这个代码加入到构造函数里，这样每个婴儿对象产生后都要执行“洗澡”代码了。

构造函数是在对象被创建时调用的方法，其声明格式如下：

```
<限定词> <类名称> ( [<参数> s])
{
  初始化程序代码
}
```

构造函数也是一个方法，不过有几点不同于一般方法的特征：

它具有与类相同的名称。

它不含返回值。

它不能在方法中用 return 语句返回一个值。

它不能被直接调用。

前面说过当在类中没有编写构造函数时，Java 编译时会自动帮我们产生一个默认构造函数，那么自动加上去的构造函数是什么样？下面以我们刚才设计的 Person 类为例：

```
Public Animal ( ) { super ( ); }
```

你可能现在不明白 super ( ) 语句的含义，后面我们会有交代。

思考一下，Earth 类有构造函数吗？

4.2　类的封装与继承

4.2.1　封装

封装是面向对象程序设计中非常重要的一个概念，也可以说是面向对象程序设计的重要特征之一。封装特性在面向对象设计中的主要贡献之一就是在接口和方法的实现上得到了分离。这样用户只要关心类的接口，而方法的实现对用户来说就变成透明的了。正因为如此，封装就成为面向对象程序设计的基础。下面通过例 4.3 说明面向对象的重要特性之一：封装。

例 4.3

```
// Eatch.java
class Person
{
  private int age;

  public Person ()
{
setAge (18);
}
```

```
  public Person (int a)
{
setAge (a);
}

  public void eat ( )
  {
      System.out.println ("Eating....");
  }

public void sleep ( )
  {
      System.out.println ("Sleeping...");
  }

  public void setAge (int a)
{
    if (a < 0 || a > 130)
    {
      System.out.println ("Wrong number");
      return;
    }
    age = a;
}

  public int getAge ( )
  {
      return age;
  }
}

public class Earth
{
    public static void main (String [] argv)
    {
Person person1 = new Person ( );
Person person2 = new Person (20);

System.out.println ("person1:"+ person1.getAge ());

// System.out.println ("person2:"+ person2.age); 编译错误
System.out.println ("person2:"+ person2.getAge ());
```

```
//person1.age = 25; 编译错误
  person1.setAge (25);
  person2.setAge (30);

System.out.println ("person1:"+ person1.getAge ());
System.out.println ("person2:"+ person2.getAge ());
person1.eat ( );
person2.sleep ( );
        }
    }
```

结果显示如图 4.3 所示。

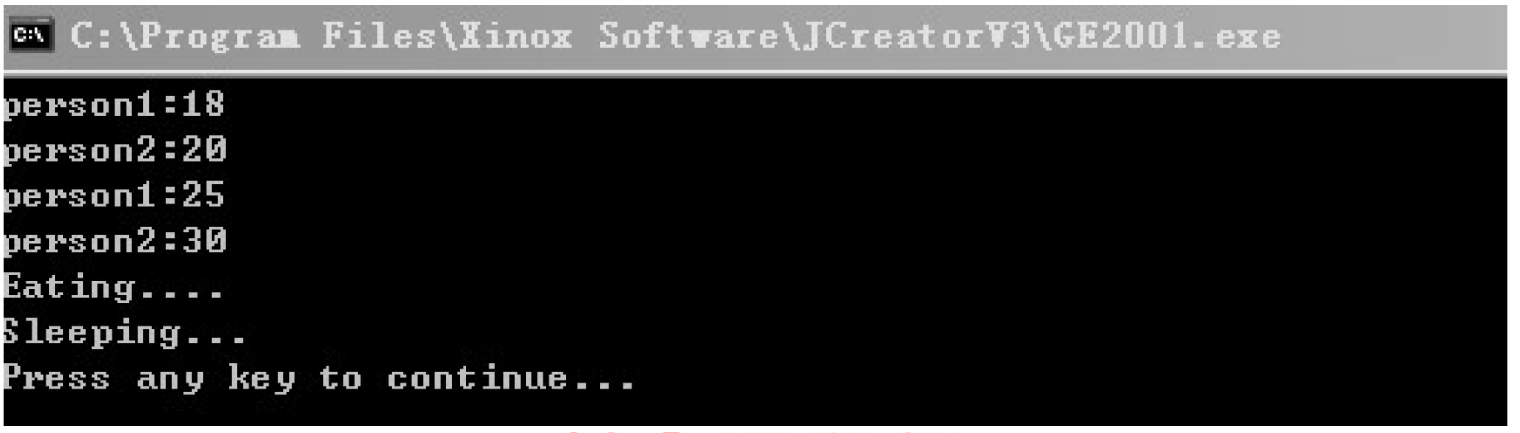


图 4.3　例 4.3 的运行结果

上面这段程序里，成员变量 age 被定义为私有（private）变量，这样就只有该类中的其他成员可以访问它，然后在该类中定义两个公有方法（public）的方法 setAge（）和 getAge（）供外部调用者访问，而 setAge（）方法可以接受一个外部调用者传入的值，当此值超出 0～130 的范围就被视为非法，就不再继续对 age 变量进行赋值操作，如果传入的值没有超出范围，就赋值给成员变量 age。通过将类的成员变量声明为私有的（private），再提供一个或多个公有（public）的方法实现对该成员变量的访问或修改，这种方法就被称为封装。

封装是面向对象程序设计中非常重要的一个特征，它除了信息隐藏作用以外，还有如下几个重要特点：

隐藏类的实现细节。对于用户来说，他不关心也不需要知道类的内部是如何运作的，只要让他知道有方法能访问这些数据就可以了。就像大家看电视，谁也不会关心电视机是如何接收节目的，只要一按遥控器，就能更换频道和收看电视节目。

强制用户使用统一的接口访问数据。这样可以方便的加入控制逻辑，限制对属性的不合理操作。

便于修改，增强代码的可维护性。

在实际应用中，对于错误赋值的处理，不能只是简单地将变量赋 0 值就算完事，可以使用更加有效的方式去处理这种非法调用，更有效地通知调用者非法调用的原因。这就是以后要讲到的抛出异常方式，这里先打个伏笔。

4.2.2 类的继承

类的继承也是面向对象的一个特色之一，它能够利用以前构造的类的方法和属性，这样会为我们节省很多的编程时间，前面我们构造了一个 Person 类，世界上各种人很多，物以类聚，人以群分，要是光靠一个 Person 类来包括许多类人，其结构性会很差的，我们必须要设计其他的类。

下面我们要设计学生（student）、老师（teacher）、工人（worker）这 3 个类，这 3 个类都继承于前面定义好的 Person 类，是 Person 类的子类，如图 4.4 所示。

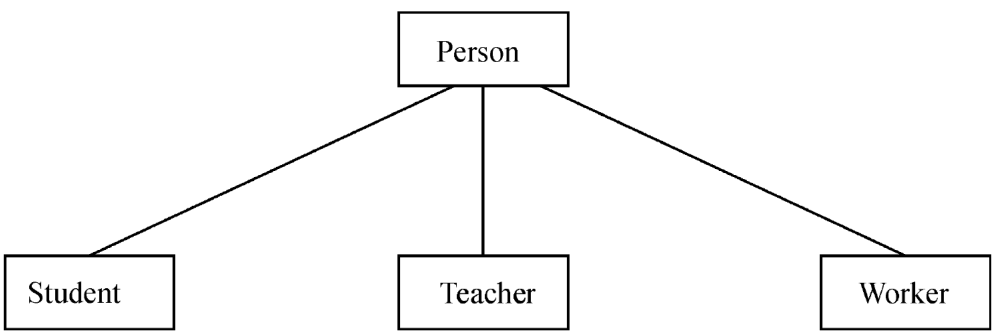


图 4.4 Person 类

下面我们来看看 Student 类的程序代码：

例 4.4

```
//Student.java
class Person
{
  private int age;
  private String name;
  private String sex;

  public Person ()
{
setAge (18);
}

  public Person (String name, String sex, int age)
{
this.age = age;
this.name = name;
this.sex = sex;
}

  public void eat ()
  {
      System.out.println ("吃饭。。");
  }
```

```
public void sleep ()
  {
      System.out.println ("睡觉。。。");
  }

  public void setAge (int age)
{
    if (age < 0 || age > 130)
    {
      System.out.println ("Wrong number") ;
      return;
    }
    this.age = age;
}
public void setName (String name)
{
      this.name = name;
}
public String getName ()
{
      return name;
}
public void setSex (String sex)
{
      this.sex = sex;
}
public String geSex ()
{
      return sex;
}

  public int getAge ()
  {
      return age;
  }
}

public class Student extends Person
{
      private String number;
      private String grade;
      public Student ()
           {
```

```
        super ("小王", "男", 21);
           this.number = "200205201220";
           this.grade = "计算机 022";
    }
     public Student (String number, String name, String sex, int age,
                     String grade)
     {
    super (name, sex, age);
    this.number = number;
    this.grade = grade;

        }

public void study ()
{
  System.out.println ("学习。。。");
}
public void ShowStudent ()
{
System.out.println ("学号:"+ this.number);
System.out.println ("姓名:"+ super.getName ());
System.out.println ("年龄:"+ super.getAge ());
System.out.println ("性别:"+ super.geSex ());
System.out.println ("班级:"+ this.getGrade ());

}

public String getNumber ()
{
  return number;
}
public void setNumber (String num)
{
  this.number = num;
}
public String getGrade ()
{
  return grade;
}
public void setGrade (String grade)
{
  this.grade = grade;
```

```
        }

    static public void main (String [] args)
    {
            Student stu1 = new Student ();
      stu1.setNumber ("200405201220");
      stu1.setName ("小仙女");
      stu1.setSex ("女");
      stu1.setAge (18);
      stu1.ShowStudent ();
      stu1.eat ();
      stu1.study ();
      stu1.sleep ();
        }
      }
```

结果显示如图 4.5 所示。

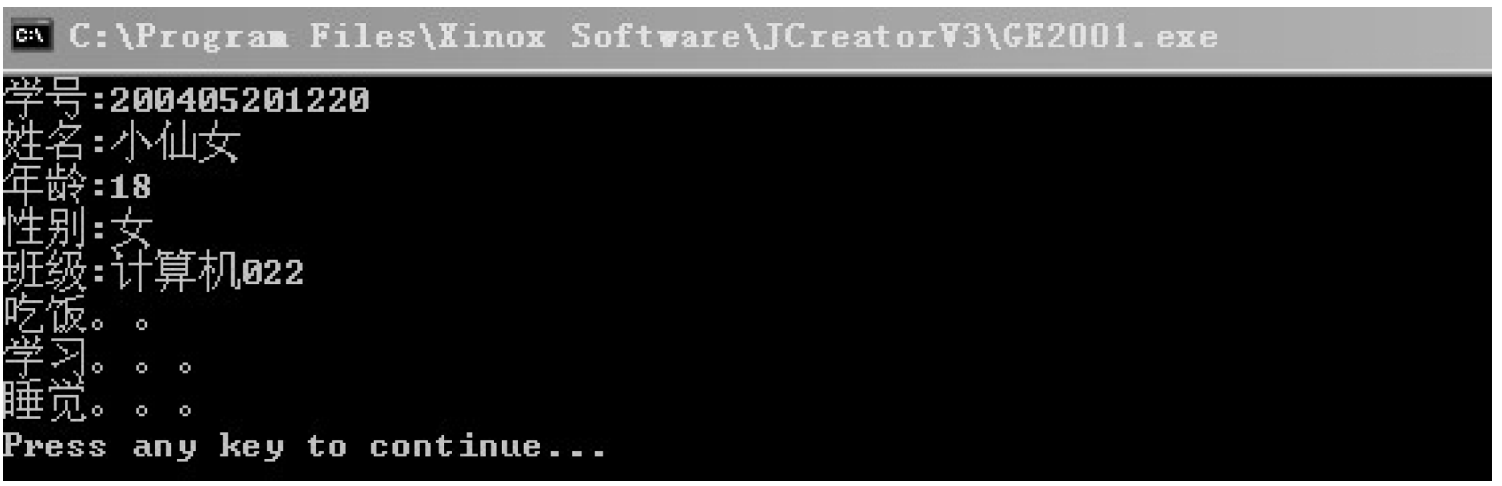


图 4.5　例 4.4 的运行结果

在上述代码中可以看到一个关键词 extends，它表明类 Student 继承了类 Person 的“所有”属性和方法，这里“所有”加了个引号，是说明这个继承并不是全盘接收，因为有些东西即使继承下来的也是不能直接使用的。例如设置为 private 的属性和方法，还有构造函数。但只要是 Person 类里面所有可继承的属性和方法，都可以在类 Student 里面使用，像 setAge（）、getAge（）、eat（）、sleep（）等这些方法都可以在类 Student 里面直接使用，上述代码中，我们在类 Student 里用了父类 Person 里的 setAge（）方法，将 age 属性赋值为 18 继承了一些东西之外，类 Student 还可以拥有一些自己的特性，例如上面代码中定义的学号 number、班级 grade，还有方法 study。

通过上述学习我们可以看到，以前已有的类中构造新类必须在新类的声明中扩展以前的类。通过扩展一个父类，可以得到这个类的一个更特殊的类，而且可以在这个基础上添加新的属性和方法。在类的继承中，有这样一些细节问题：

通过继承可以简化类的定义。

Java 只支持单继承，不允许多重继承。也就是说在 Java 中，一个子类只能有一个父类，而一个类可以被多个类继承。显然继承是一个典型的树型结构。

可以有多层继承，例如类 B 继承了类 A，类 C 又可以继承类 B，结构示意如下：

```
class A
{ }
class B extends A
{ }
class C extends B
{ }
```

4.3 多态与静态

类的多态指的是类可以以多种角色身份出现，而静态的概念恰恰相反，有时候有些成员是所有对象共有的，只需要在内存中保存一份就可以了。

4.3.1 类的多态

下面我们在形状类 CShape 基础之上，再设计一个子类椭圆 Ellipse，子类矩形类 CRectangleEx，以及矩形的一个子类有圆角的矩形（RectangleEx）类层次结构图，如图 4.6所示。

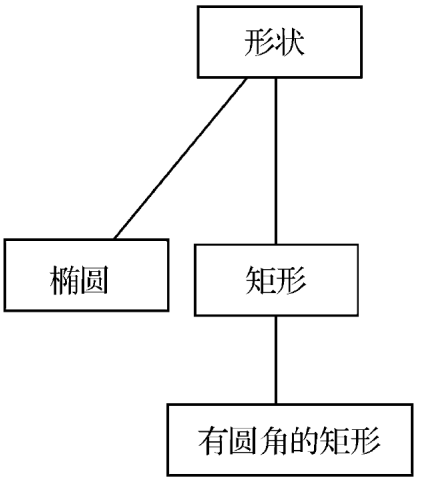


图 4.6 层次结构图

例 4.5

```
// CShape.java
package shapePackage;
import java.awt.*;
public class CShape
{
    protected Color color;
    protected int lineSize;
        public CShape ()
         {
          color = Color.black;
          lineSize = 1;
               }
public CShape (Color c, int size)
{
  this.color = c;
  this.lineSize = size;

}
public void setColor (Color c)
{
  this.color = c;
}
public Color getColor ()
```

```
        {
          return this.color;
        }
        public void setLineSize (int size)
        {
          this.lineSize = size;
        }
        public int getLineSize ()
        {
          return this.lineSize;
        }
        public void draw ()
        {
        System.out.println ("画图形 .....");

            }
        }
//Ellipse.java
package shapePackage;
public class Ellipse extends CShape
{
        int centerX;
        int centerY;
        int width;
        int height;
        public Ellipse (int x, int y, int w, int h)
        {
              centerX = x;
              centerY = y;
              width = w;
              height = h;

        }
        public void draw ()
        {
              System.out.println ("画椭圆 ..");
        }

}
//CRectangle.java
package shapePackage;
public class CRectangle extends CShape
{
```

```
        int left;
        int top;
        int width;
        int height;
        public CRectangle (int l, int t, int w, int h)
         {
              left = l;
              top = t;
              width = w;
              height = h;
        }
        public void draw ()
        {
              System.out.println ("画矩形 ...");

        }

}
// RectangleEX.java
package shapePackage;
public class RectangleEx extends CRectangle
{
        int radius;
        public RectangleEx (int l, int t, int w, int h, int r)
        {
        super (l, t, w, h);
        radius = r;

    }

    public void draw ()
    {
        System.out.println ("有圆角的矩形 ...");
    }
    }
// MainClass.java
package shapePackage;
import java.awt.*;
public class MainClass
{
        static public void main (String [] args)
   {
        CShape [] objShapes = new CShape [4];
```

```
            objShapes [0] = new CShape (Color.blue, 3);
            objShapes [1] = new Ellipse (30, 40, 50, 50);
            objShapes [2] = new CRectangle (30, 40, 50, 50);
            objShapes [3] = new RectangleEx (30, 40, 50, 50, 30);
        //画图形
        objShapes [0] .draw ();
        //画椭圆
        objShapes [1] .draw ();
        //画矩形
        objShapes [2] .draw ();
        //有圆角的矩形
        objShapes [3] .draw ();

        }
        }
```

结果显示如图 4.7 所示。

图 4.7　例 4.5 的运行结果

除前面讲到的封装和继承外，面向对象的程序设计还有一种基本特征就是多态。

当类从某个父类继承时，可能需要修改父类中某个方法的实现方式，多态机制能够分别实现是用于不同的子类的方法，但方法的定义在共同的父类中进行。当子类中的方法被调用时，启动的是该子类的方法。尽管该方法的实现在各个子类之间是不同的，但是对于外部调用来说，不必关心具体的实现，只需关心其共同的接口。

多态机制是面向对象技术的精华之一，它是建立在继承基础之上的。当方法被不同的对象调用时，能产生不同的行为，这种现象就称为多态。

例 4.5 中，不管是椭圆、矩形、还是圆角的矩形，都有绘制操作，对应的方法为 draw。但是它们的绘制都不相同，利用面向对象的多态机制，可以在父类 CShape 中定义该方法，分别在子类中去实现。于是，需要在父类中添加一个方法声明：

```
public void draw ()
        {
        System.out.println ("画图形……");

        }
```

Draw 方法在子类中已经分别实现，这样，就利用多态实现了不同的对象产生不同的行为，虽然调用的是同一个方法。

类之间除了强制转换外，也有自动转换，原则如下：

父类转换成子类属于强制转换。

子类转换成父类属于自动转换。

兄弟类之间（如 Worker 类和 Teacher 类）可以用强制转换的方式来转换。

从上面我们清楚了类可以是一个“变色龙”，不同场合下可以有不同的角色身份出现，那么有的时候我们需要确切的知道，这个对象到底是属于那个类的？Java 中为我们提供了一个识别运算符“instanceof”，它可以用来判断 A 对象是不是 B 类的实例，格式如下：

```
<对象名称> instanceof <类名称>
```

它的返回值是布尔型的，真（true）、或假（false）。

例如下面比较的结果是 true：

```
CShape d1 = new Ellipse (30, 40, 50, 50) (;
d1 instanceof Ellipse;
```

想想看，一个司机是属于工人吧，同样也属于人。而下面的比较结果是 false：

```
Ellipse w1 = new Ellipse (30, 40, 50, 50);
w1 instanceof CShape ();
```

一个工人是不能把他当司机看的，这和现实生活中是一致的。

4.3.2 静态成员的定义与使用

在介绍静态成员的定义与使用方法之前先看一个例子。

例 4.6

```
//MainClass.java
class Student
{
        private String Snum;
        private String Sname;
        private int Sex;

        static private int count;
              static
              {
              count = 0;
              }
        static public int getCount ()
               {
                    return count;
                }
        public Student (String num, String name, int sex)
           {
              this.Snum = num;
              count ++ ;
```

```
            this.Sex = sex;
            this.Sname = name;
          }
        public void ShowStudent ()
         {
          System.out.println ("学号:"+ this.Snum);
          System.out.println ("姓名:"+ this.Sname);
          System.out.println ("性别:"+ this.Sex);

        }
        public void study ( )
        {
          System.out.println ("学习中。。");
        }
    //……
    //还有其他的属性和方法已省略
}
public class MainClass
{
        public static void main (String argv [ ])
        {
        //第一个学生
         Student obj = new Student ("s001","李世明", 29);
         obj.ShowStudent ();
         System.out.println ("学生总数:"+ Student.getCount ());
        //  第二个学生
         Student obj2 = new Student ("s002","李治", 19);
         obj2.ShowStudent ();
         System.out.println ("学生总数:"+ Student.getCount ());
        //第三个学生
         Student obj3 = new Student ("s003","陈晓强", 29);
           obj3.ShowStudent ();
         System.out.println ("学生总数:"+ Student.getCount ());
         }
    }
```


程序运行结果如图 4.8 所示。

上面这个例子是非常好的一个例子，读者可以好好领会例子。

类成员的调用格式如下：

<类名称>.<类成员名称>

类的静态成员经常被称为“类成员”，对于静态成员变量，我们叫类属性，对于静态成员方法，我们叫类方法。

还有一点需要说明的是，在静态方法里只能直接调用同类中其他的静态成员（包

括变量和方法)，而不能直接访问类中的非静态成员。这是因为，对于非静态的方法和变量，需要先创建类的实例对象后才可使用，而静态方法在使用前不用创建任何对象。

图 4.8　例 4.6 的运行结果

4.4　面向对象的基本概念

其实对象这个词对于我们来说并不陌生。在我们身边处处都有对象，小到一个细菌，大到一个星球，几乎所有的东西都可以看成为一个对象，在这一小节里，我们会以深入浅出的方式，引导你了解什么是对象及其相关概念。

4.4.1　类与对象

对象是理解面向对象技术的关键。我们先来看看平常是如何来称呼一些对象的。例如你看到小王买了一辆奔驰 990 汽车，你会说小王家买了一辆汽车，他们家真有钱。你经常会说天空有许多鸟，我想买一件衣服，昨天丢了一个手机，屋角有只狗等。可以看出，我们通常并不是直接称呼对象本身的，而是以这个对象的分类的名称来称呼的。上面所说的汽车、鸟、衣服、手机、狗等，都只不过是一种对象的分类而已，而这些分类在面向对象世界里我们习惯称为“ 类”。现实生活中有各式各样形形色色的类，比如说汽车类、鸟类、手机类等。类是某一类事务的描述，是抽象的、概念上的定义。

对象是实际存在的该类事务的个体，是由类定义所产生出来的实例（instance)。虽然我们用类的名称来称呼这些对象，但实际上看到的还是对象本身的实例。例如，汽车设计图就是“ 类”，由这个图纸设计出来的若干汽车就是按照该类产生的对象。可见类是对象的模板。一个类可以对应多个对象。类只是个抽象的称呼，而对象是个看得到、摸得到、听得到的实例。所以面向对象程序设计的重点是类的设计，而不是对象的设计。

同一个类按照相同的方法可以产生出许多个对象，刚开始时这些对象的状态都是一样的，因为是一个模子上生产出来的，就像按照一张手机图纸生产出来的手机都是一模一样的。如果有个用户买回去后，把壳子改装了一下，这只会使得这个用户的手机发生变化，而不会影响到其他相同型号的手机。但如果把手机图纸修改了，那就会

使得以后生产的手机发生变化。

4.4.2　成员

而有的时候为了区分对象与另一个对象，我们直接使用对象的名称来称呼，例如有个同学类，同学类里有个男同学叫张猛（具体的对象），还有个女同学叫李柔（具体的对象），张猛和李柔同属于同学这个类，但他们之间却不相同，因为他们的名字不同，性别不一样，身高体重都不相同。像姓名、性别、身高、体重这些东西都可用来描述同学的特征，我们把它称之为“属性”(attribute)。由此可以看出属性是用来形容一个实例对象的，也因为有了这些属性，世界上每个对象都不同，就算同学类里恰巧有两个同名的李柔，我们也可以通过其他特征辨别出来此李柔非彼李柔也，例如身高，可能一个李柔高而另一个矮，眼睛可能是一个是双眼皮而另一个是单眼皮，就算是双胞胎，也有不一样的地方。

上面的属性描述了对象的静态方面的特征，但对象也有一些动态的行为，我们把这些行为称之为“方法”(method)，通过使用这些方法，可以让对象执行一些操作。例如人里面如果定义了一个方法叫“喝酒”，那么我们就可以调用人里定义的方法“喝酒”，去执行喝酒的过程。

说了属性也谈了方法，我们把属性和方法称为这个对象的“成员”，因为它们构成了一个对象的主要部分，有了它们才使得一个对象栩栩如生，它们充分说明和描述了一个对象。没有了属性和方法，对象就变空了，没有了存在的意义。人活在这个世界上是要有意义的，同样对象也一样，没有了属性和方法，对象就不存在了。

我们已经知道类和对象，一个抽象一个具体，类可以说是对象的“模子”，这个模子定义了对象应该具有的一些“音容笑貌”，也就是它的属性和方法。因此通过类生产出来的对象，整个框架都是一样的，有多少属性那些方法，都是一样的。所以使得我们很容易辨别出哪一个对象是属于什么类，进而知道这个对象有什么样的行为。

4.4.3　继承

继承？类也可以有子女吗？类的子女之间也争夺财产吗？子类如何继承父类？看来面向对象的许多特性都来源于现实生活，这样才使得计算机描述的世界与现实世界高度一致。继承是一种机制，可以让一个类能够继承另一个类的所有行为和属性。因此通过继承，一个类可以拥有已有类的所有功能，所以只需要指明新类与现有的类不同就可以了。继承其他类的类叫做子类，被继承的类叫做父类。看来子类是青出于蓝而胜于蓝，如果子类与父类一模一样没有新意的话，定义子类是一点意义也没有的。所以继承最主要的目的是为了扩展原类的功能、加强或改进原类所没有定义的属性和方法。例如有一个学生类，而学生可分为小学生、中学生、大学生、成人学生等，如果就用一个学生类来涵盖所有各类学生的属性和方法，这是不可能的，像身份证号码这一属性，在小学生里就没有，但在大学生里和成人学生里都有。所以我们可以把学生中的一些共有的属性和行为提取出来，形成学生这个类，然后在学生这个类下面再定义一些子类，如：小学生类、中学生类、大学生类、成人学生类等。这些子类会继承学生类中所有公共的可以继承的属性和方法，然后再加上一些自己这个类所特有的

属性和方法，或者还可以修改原本不适用于这个类的方法，如图 4.9 所示。

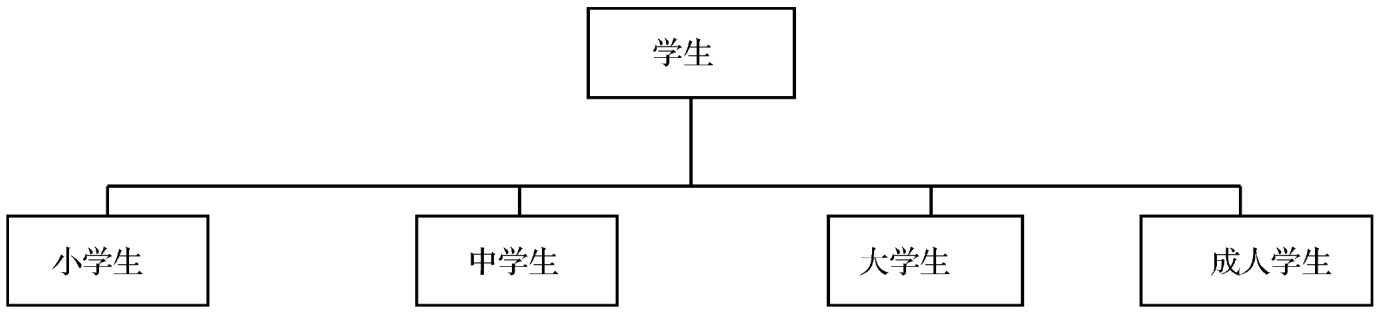


图 4.9　学生类及其子类

4.4.4　多态

多态就是多种形式的意思。例如前面说过的男同学张猛，可以说他是个人，还可以说他是个男人，还可以说他是个男同学。也就是说张猛可以以多种身份出现，但这些身份都是继承的关系，如男人类是人类的子类，男同学类是男人类的子类。我们经常可以用一个类的上层类来说它，这都是可以的。

多态是面向对象程序设计中的另外一个非常重要的概念。简单地说，多态就是一个类有其他的表示方式，但是彼此之间必须是继承的关系。例如碰到张猛，我可以给旁边的人说他的闲话，“这个男同学如何如何”，也可以说“这个男人如何如何”，还可以说“这个人如何如何”，说的人和听的人都知道我说得是张猛。

使用多态概念时，有 3 点必须要注意到的：

1）张猛就是张猛，他不会因为不同的称呼就改变了它原来的实例。张猛是男同学类所产生的一个对象，而不是一个人类的对象。

2）当你把张猛当成男人看时，你只能使用和访问男人类所提供的属性和方法。而不能用男同学类下所特有的属性与方法。因为你把他当男人来看，就看不到他男同学的一面，所以男同学类下所特有的属性与方法也用不了。也就是说当以父类的身份来看时，子类中特有的属性与方法是不能用的。

3）如果父类有方法被子类覆盖（override）时，那么当你以父类的身份来调用这个方法时，将会执行子类的方法。是不是觉得与第二点有矛盾，仔细读一下，第二点说得是父类调用子类中特有的属性与方法，而本条说得是父类调用子类中与父类同名的方法。

4.5　重载、屏蔽与覆盖

4.5.1　重载

在 C 语言中对于不同类型的两数运算，只要类型不同就得写几个函数，而且函数名字不能相同。但是 Java 语言的编程思想更接近我们的思维，可以将这些函数名取同样的名字，这种机制我们称之为重载。下面用例 4.7 说明重载。

例 4.7

```
// MainClass.java
```

```
public class MainClass
{
      static public int Add (int a, int b)
     {
           return a + b;
      }
      static public float Add (float a, float b)
     {
           return a + b;
      }
      static public int Add (int a, int b, int c)
     {
           return a + b + c;
      }
      static public float Add (float a, float b, float c)
     {
      return a + b + c;
      }
static public void main (String [] args )
     {
           int a, b, c;
           float f1, f2, f3;
           f1 = 5; f2 = 8; f3 = 3;
           a = 3; b = 4; c = 5;
           System.out.println ("int a + b = "+ Add (a, b));
           System.out.println ("int a + b + c = "+ Add (a, b, c));
           System.out.println ("float f1 + f2 = "+ Add (f1, f2));
           System.out.println ("float f1 + f2 + f3 = "+ Add (f1, f2, f3));
           }
   }
```


显示结果如图 4.10 所示。

图 4.10　例 4.7 的运行结果

前面我们已经学过什么叫构造函数的重载，由于构造函数也是方法的一种，所以对于一般方法来说，也是有重载这个机制的。

在某些情况下，可能要在同一个类中写几种带有不同参数但做同样工作的方法。如例子中设计了一个求和方法 Add ()。现在假设要求 int、float 中同类型的两个或三个数相加的方法，这是合情合理的，因为各种数据类型要求不同的格式，而且可能要求不同的处理。那么我们可以分别创建 4 种不同的方法，即：AddInt1 ()、AddInt2 () 和 AddFloat1 ()、AddFloat2 () 但这是很繁琐的。

Java 允许对不止一种方法重用方法名称。那么上述三种不同的方法可以取相同的名称，但是调用时能区分开来到底调用的是哪个方法呢？它们是在参数的数量和类型的基础上对此进行区分。通过重用方法名称，可用下述方法：

```
public int Add (int a, int b);
public int Add (int a, int b, int c);
public float Add (float a, float b);
public float Add (float a, float b, float c);
```

当写代码来调用这些方法中的一种方法时，根据提供的参数的类型选择合适的一种方法。重载遵守的规则如下：

方法名称一定要一样，不一样的话，就是两个不同的方法，不能称为重载。

传递的参数类型不一样或参数个数不同。因为编译器要通过参数类型或参数个数来判断调用哪一个方法。

4.5.2 屏蔽

关于屏蔽，首先来看例 4.8。

例 4.8

```
public class ExampleTest
{
  public static void main (String argv [])
  {
    A a = new A ( );
    B b = new B ( );
    System.out.println ("a.i = "+ a.i);
    System.out.println ("b.i = "+ b.i);
  }
}
class A
{
  int i = 10;
  public A ( )
  {
    System.out.println (i);
  }
}
class B extends A
{
```

```
    public B ( )
    {
      System.out.println (i);
    }
  }
```

程序运行结果如图 4.11 所示。

图 4.11　例 4.8 的运行结果

例 4.8 中 B 类只是单纯地继承了 A 类，所以在 A 和 B 的构造函数中显示出的 i 值都是 10。如果 B 类自己也增加一个 i 的属性，例如：

```
class B extends A
{
  int i =5;
  public B ( )
  {
    System.out.println (i);
  }
}
```


改造后程序运行，b.i 就会显示出 5，使用的是自己的 i。

下面需要说明的是，如果想要在子类中使用父类的属性，应该如何表达呢？我们可以用 super 这个关键字，格式如下：

```
super. < 属性名称 >
```

所以如果我们在 B 类中要使用 A 类的 i 属性，可以使用 super.i 的方式来访问。

那么什么是屏蔽？屏蔽其实就是继承当中，在子类里可以直接使用从父类继承下来的属性和方法，但如果在子类中又声明了相同名称的属性的话，那么直接使用时，用的就是子类的属性，而不是继承自父类的属性了，这种情形我们称之为“屏蔽”。

4.5.3　覆盖

父类的属性会被子类中相同名称的属性屏蔽，那么对于方法来说也是一样的，不过不用屏蔽，而是用覆盖这个词。屏蔽指的是被盖着看不到，而覆盖是真正地改写了原本父类的方法。

覆盖中子类可以修改从父类继承来的方法，但具有相同的名称、返回类型、参数表，下面看一个例子。

例 4.9

```
class Employee
{
  protected String name;
  protected int salary;
  public Employee (String n, int s)
  {
      this.name = n;
      this.salary = s;
  }
  public String getDetails ( )
  {
    return "Name: " + name + "\n" + "Salary: " + salary;
  }
}
class Manager extends Employee
{
  String department;
  public Manager (String name, int s, String d)
  {super (name, s);
  this.department = d;
  }
  public String getDetails ( )
{
      return "Name: " + name + "\n" + "Manager of " + department;
  }
}
public class MainClass
{
      static public void main (String [] args)
      {
            Employee obj = new Employee ("李世明", 1000);
            Manager obj2 = new Manager ("李治", 800, "浙林院");
            System.out.println ("调用父类 Employee 的对象 obj 方法 getDetails");
            System.out.println (obj.getDetails ());
            System.out.println ("调用子类 Manager 的对象 obj2 方法 getDetails");
            System.out.println (obj2.getDetails ());
      }
}
```


显示结果如图 4.12 **所示。**

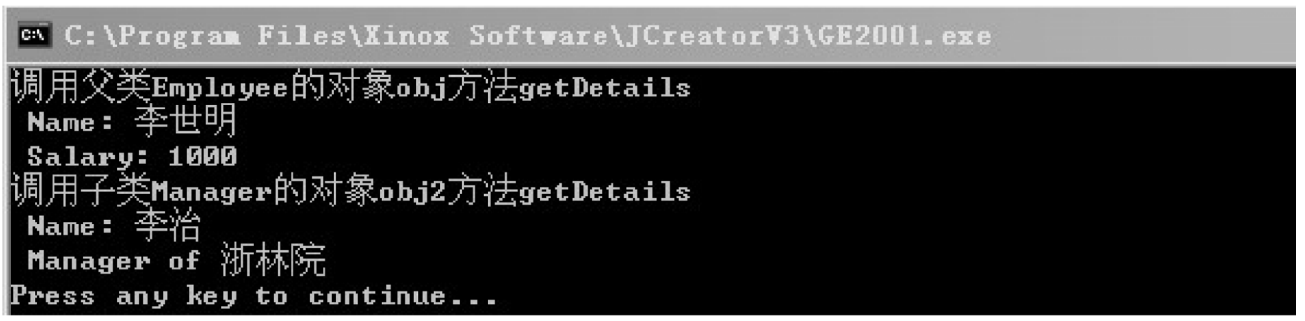


图 4.12　例 4.9 的运行结果

Manager 类有一个定义的 getDetails（）方法，因为它是从 Employee 类中继承的，原来的方法在子类中被覆盖了。

请求 e.getDetails（）和 m.getDetails 就会有不同的结果。Employee 对象将执行与 Employee 有关的 getDetails（），Manager 对象将执行与 Manager 有关的 getDetails（）。如果执行下述语句，那么是执行 Employee 中的 getDetails（），还是 Manager 中的 getDetails（）呢？

```
Employee obj3 = new Manager ();
Obj3.getDetails ();
```

我们说将会执行的是 Manager 中的 getDetails（）。从多态角度来看这也是合理的。只有一种情况下会调用父类的方法，就是使用 super 调用，如果在 Manager 类中使用：

```
super. getDetails ( );
```

那么将会调用 Employee 中的 getDetails（），因为 super 代表父类的意思，所以这样的调用有这样的结果并不意外。

方法覆盖在使用上也有些规定要遵守：

方法名称一定要一样。不一样的话，就是两个不同的方法，不能称为覆盖。

返回值的数据类型要一样。不一样的话，就是重载而不是覆盖。

使用的参数要一样，包括参数个数及每个参数的数据类型也要一样，不一样的话就是重载而不是覆盖。

小　　结

本章主要讨论了：

构造函数的使用。

多态和封装的概念与使用方式。

继承的概念。

类成员和实例成员之间使用上的差异。

重载、覆盖与屏蔽的使用规则。

什么是类？什么是对象？什么是封装？什么是多态？

习　　题

1. 下面的类中，哪些是合法的重载？

```
Public class Q1
{
    public void method (int i) { };
}
```

A. private void method (int i) { }　　B. public void method (int k) { }

C. public int method (int i) { }　　D. private float method (float f) { }

E. public String method (int i, int j) {}

2. 在 Q2_2 类中，哪些是合法的覆盖？

```
public class Q2_1
{
    public void method (int I) { };
}
class Q2_2 extends Q2_1 { }
```

A. public void method (int i) { }　　B. private void method (int j) { }

C. public int method (int i) { }　　D. public float method (float f) { }

E. private String method (String s) { }

3. 同上题，哪些 method 属于重载？

4. Q4_2 类运行的结果是什么？

```
public class Q4_1
{
    public void method (int i)
    {
        Systems.out.println (10);
    }
}
public class Q4_2 extends Q4_1
{
    public static void main ( String argv [])
    {
      Q4_1 q1 = new Q4_1 ( );
      Q4_2 q2 = new Q4_2 ( );
      q1.method (1);
      q2.method (2);
    }
    public void method (int k )
    {
          Systmes.out.println (20);
```

```
        }
    }
```

A.10，20　　B.10，10　　C.20，20　　D.1，2　　E.20，10

5. 承上题，若把 Q4_2 类的第 6 行改成

```
Q4_1 q2 = new Q4_2 ();
```

则最后运行的结果是什么？

A.10，20　　B.10，10　　C.20，20　　D.20，10　　E.1，2

6. 承第 4 题，若把 Q4_2 类的第 8 行改成

```
((Q4_2) q2).method();
```

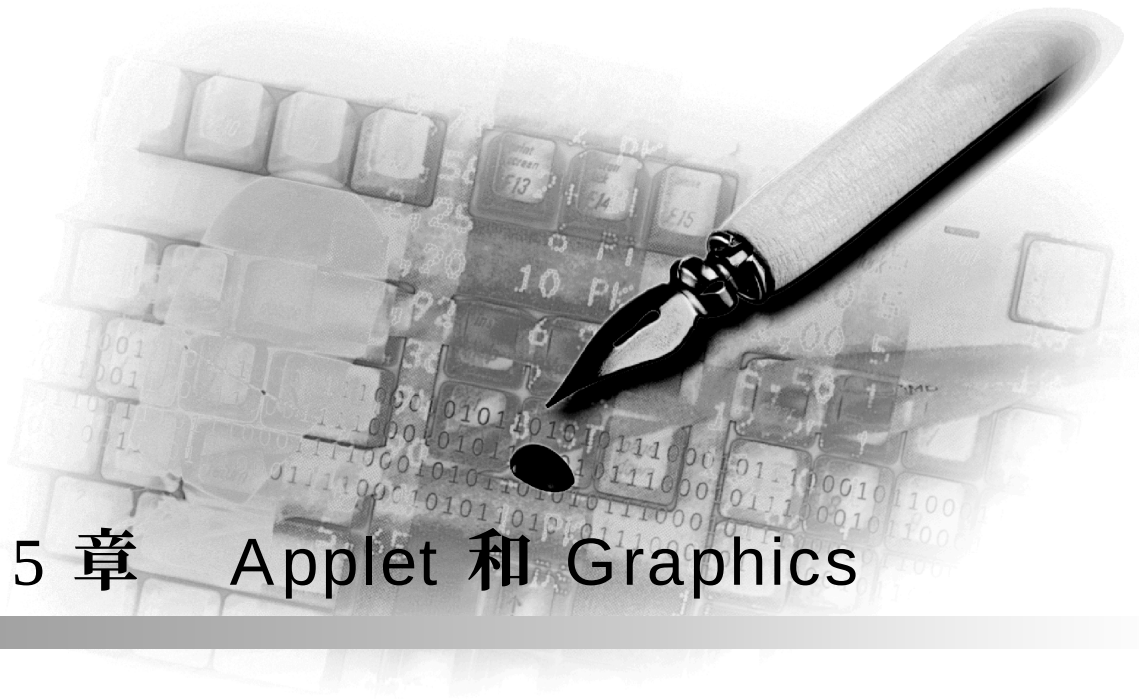
最后运行结果是什么？

A.10，20　　B.10，10　　C.20，20　　D.20，10　　E.1，2

7. 请将例 4.4 中类 Person 的 Teacher 子类和 Worker 子类代码按照自己的设计写出来，再设计一个 Worker 类的子类。

8. 屏蔽与覆盖有什么区别？请编程说明。

9. 简述一下封装、继承和多态的关系。



第 5 章　Applet 和 Graphics

本章要点

Java.applet.Applet 类
Applet 的生命周期
Applet 应用程序
applet 标记和 HTML
使用 Graphics 类绘制图形
定义 Circle 类

本章学习目标

掌握 Applet 程序与 Application 程序的区别
了解 Applet 类的层次
掌握 Applet 的生命周期和运行方法
掌握 Applet 标记的语法
掌握利用 Graphics 类在屏幕上绘制文本、线条、椭圆形和弧形等的对象

在 Java 中，有两种类型的程序：应用程序（Application）和小应用程序（Applet）。Applet 是用 Java 编写的小应用程序，它能够嵌入在 HTML 网页中，之所以如此热门，其根本原因在于 Java 具有“让 Internet 动起来”的魅力，可以由支持 Java 的 Web 浏览器来解释执行。Java 语言之所以能够在那么短的时间内迅速普及，这和 Applet 的功劳是不能分开的。

Applet 是 Java 中一个十分重要的部分，本章将结合实例讲述 Applet 的工作原理以及如何编写 Applet 程序。

5.1　简单的 Applet

一个 Applet 对象有很多变量和方法，大部分来自于 JDK 中关于 Applet 的定义，当你调入 java.applet.Applet 时，你就能了解到它的定义。大多数的情况下，也会调入 java.awt.*，这个包含很多对视窗和图形有用的类。下面是一个小的 Applet 的代码，类名为 Test。

5.1.1　编辑 Applet

下面是一个简单的名为 Test 的 Applet 代码。

例 5.1

```
import java.applet.Applet;          //引入 Applet 类
import java.awt.*;
public class Test extends Applet  // 定义一个名为 Test 的类
{
  public void paint ( Graphics gr )
  {
    setBackground ( Color.pink );
    gr.drawString ("          黄鹤楼                ", 25, 30);
    gr.drawString ("昔人已乘黄鹤去，此地空余黄鹤楼。", 25, 50);
    gr.drawString ("黄鹤一去不复返，白云千载空悠悠。", 25, 70);
    gr.drawString ("晴川历历汉阳树，芳草萋萋鹦鹉洲。", 25, 90 );
    gr.drawString ("日暮乡关何处是，烟波江上使人愁。", 25, 110);
    gr.drawString ("---崔颢", 50, 150);
  }
}
```


程序中首先通过 import 语句引入类 java.applet.Applet 和 java.awt.*（关于类的知识将在本书的下节中详细介绍）。然后通过语句

```
public class Test extends Applet
```

定义一个名为 Test 的类，该类继承了 java.applet.Applet 类。接着重写了方法 paint，在这个程序中，它负责下面的一系列字符串写到 Applet 容器指定位置。

5.1.2　编译 Applet

源文件编辑好后存盘，文件名必须与类名保持一致，即 Test.java。然后打开一个窗

口，进行编译。在命令行键入：

```
C:\code >javac Test.java
```

编译的结果是生成字节码文件 Test.class。

5.1.3 运行 Applet

为了运行该程序，还需要将其字节码文件嵌入到一个 HTML 文件，然后用浏览器或者 appletviewer 打开 HTML 文件执行。例如，打开编辑器输入名为 Cui.html 的文件源代码并把这些文件与 Test.class 放在同一目录。

```
<html>
<head>
<title>test</title>
</head>
<body>
<applet code="Test.class" height="200" width="300"></applet>
</body>
</html>
```

在命令行键入：

```
C:\code>appletviewer Test.html
```

这时屏幕上弹出一个窗口，显示运行结果。具体运行如图 5.1 所示，该例在 appletviewer中的运行结果如图 5.2 所示。

图 5.1 程序 Test.java 的编译及运行方式

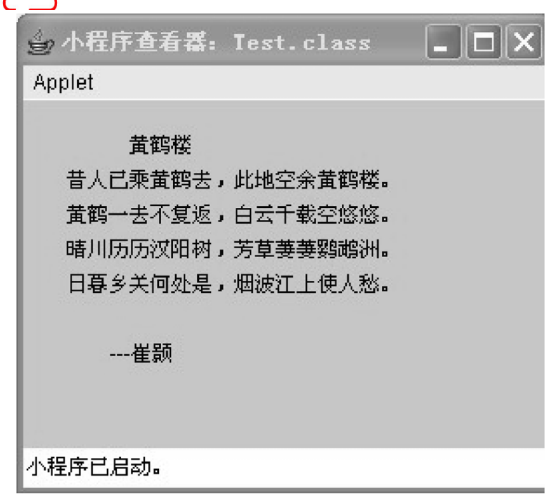


图 5.2 例 5.1 的运行结果

从上面我们看到，Applet 是使用 Java 语言编写的一段代码，它能够在浏览器环境中运行。与 Application 的区别主要在于其执行方式的不同。Application 是从其中的 main ( ) 方法开始运行的，而 Applet 是在浏览器中运行的，必须创建一个 HTML 文件，通过编写 HTML 代码告诉浏览器载入何种 Applet 以及如何运行它。

Applet 程序也可以直接利用 appletviewer 提供的图形用户界面，而 Application 程序则必须另外书写专用代码来构建自己的图形界面。

5.2　Applet 类的层次

要编写一个 Applet，必须创建一个类：

```
import java.applet.*;
import java.awt.*;
public class Test extends Applet
{
…
}
```

利用 import 指令，将所有 java.applet.* 和 java.awt.* 的类函数加入到这个 Applet 程序里。Applet 的类必须为 public，且它的名称必须与它所在的文件名匹配，在这里，就是 Test.java。而且，该类必须为 java.applet.Applet 的子类。

Applet 的基础类是 java.applet.Applet 类，它扩充自 java.awt.Panel 类，所以可以说 Applet 是一些面板（Panel）。而 java.awt.Panel 类又扩充自 java.awt.Container 类，所以也可以认为 Applet 是一些容器（Container）。再往下看，会发现 java.awt.Container 类扩充自 java.awt.Component 类，所以又可以说 Applet 是一些构件（Component），这也意味着 Applet 有能力处理各种事件，并能被添加到各种容器中。图 5.3 是 Applet 的类层次结构。

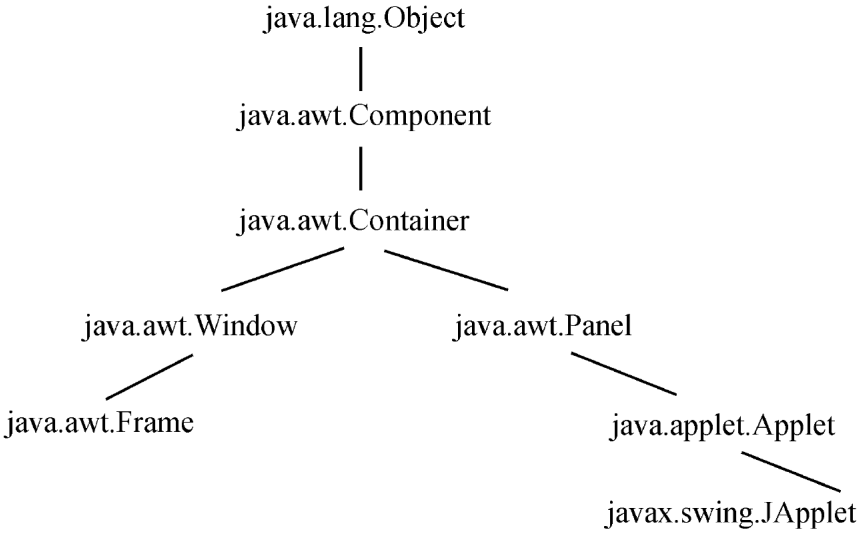


图 5.3　Applet 的类层次结构

Applet 类是 Swing JApplet 的超类，所以如果想使用 Swing 集合来实现 JApplet，那么编制的 JApplet 应该继承 Applet 类。

5.3　Applet 的生命周期

5.3.1　Applet 的生命周期举例

下面的例子使用了 Applet 生命周期中的这几个方法。

例 5.2

```
import java.awt.Graphics;
```

```
import java.applet.Applet;
public class HWloop extends Applet
{
AudioClip sound;     // 声音片断对象
public void init ()
{
sound = getAudioClip ("hello.au");     // 获得声音片断
}
public void paint (Graphics g)
{
g.drawString ("hello Audio", 25, 25);// 显示字符串
}
    public void start ()
    {
      sound.loop ();// 声音片断开始播放
    }
    public void stop ()
    {
      sound.stop ();// 声音片断停止
    }
}
```


在本例中，Applet 开始执行后就不停的播放声音，如果浏览器图标化或者是转到其他页面，则声音停止播放；如果浏览器恢复正常或者是从其他页面跳回，则程序继续播放声音。

5.3.2 Applet 的方法

Applet 的生命周期相对于 Application 而言较为复杂。在 Applet 的生命周期中涉及到 Applet 类的 4 个方法（也被 JApplet 类继承）：init ()、start ()、stop () 和 destroy ()，它们构造了创建任何 Applet 的框架。

在实际应用中，用户需要重载这些方法来构造自己需要的 Applet。

1. init ()

当 Applet 第一次被支持 Java 的浏览器加载时，该方法会被自动调用。在 Applet 的生命周期中，只执行一次该方法，因此可以在其中进行一些只执行一次的初始化操作，如处理由浏览器传递进来的参数、添加用户接口组件、加载图像和声音文件等。

2. start ()

系统在调用完 init ()方法之后，将自动调用 start ()方法。当用户刷新包含 Applet 的页面或者从其他页面返回包含 Applet 的页面时，start ()方法也会被自动调用。也就是说，start ()方法可以被多次调用，这与 init ()方法是有区别的。基于这样的原因，可以把只调用一次的代码放在 init ()方法中，而不能放在 start ()方法中。start () 方法的典

型用法是启动动画和播放声音。

3.stop ( )

当用户离开包含 Applet 的页面时会自动调用 stop ( )方法。和 start ( )方法一样，stop ( )方法也可以被多次调用。该方法的作用是，当用户离开包含 Applet 的页面时，停止一些耗费系统资源的活动，如播放动画等。如果在 Applet 中没有动画、音乐文件播放，通常也不必重载该方法。

4.destroy ( )

浏览器正常关闭时，Java 自动调用这个方法。destroy ( ) 方法用于回收任何一个与系统无关的内存资源。如回收图形用户界面的系统资源、关闭连接等。至于 Applet 实例本身，会由浏览器来负责从内存中清除，不需要在 destroy ( )方法中来清除。

5.3.3　Applet 的生命周期和对应的方法

Applet 的生命周期中有 4 个状态：初始态、运行态、停止态和消亡态。当程序执行完 init ( )方法以后，Applet 程序就进入了初始态。然后马上执行 start ( )方法，Applet 程序进入运行态。当 Applet 程序所在的浏览器图标化或者是转入其他页面时，该 Applet 程序马上执行 stop ( )方法，Applet 程序进入停止态。在停止态中，如果浏览器又重新装载该 Applet 程序所在的页面，或者是浏览器从图标中复原，则 Applet 程序马上调用 start ( )方法，进入运行态。当然，在停止态时，如果浏览器关闭，则 Applet 程序调用 destroy ( )方法，进入消亡态。其过程如图 5.4 所示。

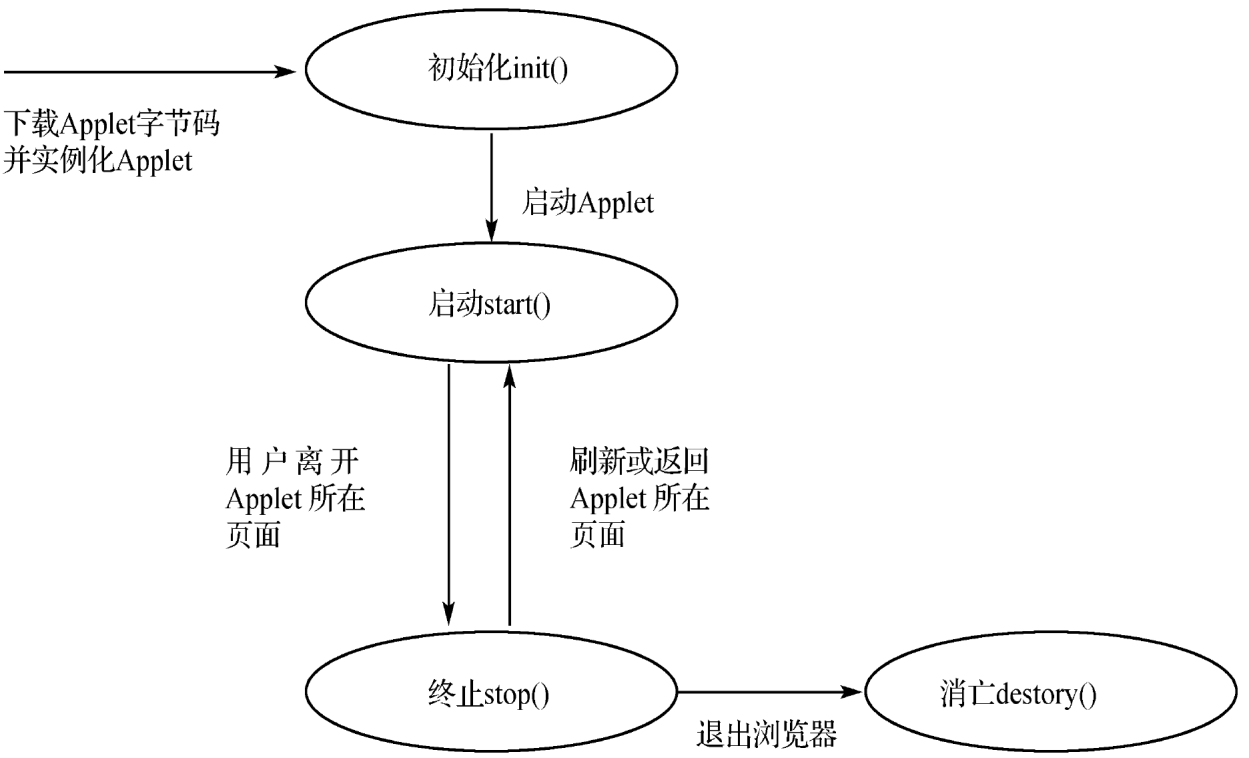


图 5.4　Applet 的生命周期和对应的方法

5.4 Applet 标记和 HTML

本书前面已经提过，Applet 不能独立运行，为了运行该程序，还需要将其字节码文件嵌入到一个 HTML 文件中，然后由浏览器解释执行。

下面是使用 Applet 标记的最简单形式。

```
<applet code = "HWloop.class" width = 200 height = 100 >
</ applet >
```

其中 code 是一个必需的属性，它给定了需要运行的 Applet 的文件名，并且必须包含扩展名 .class。width 和 height 也是一个必需的属性，指定了容纳该 Applet 的窗口大小。

5.4.1 Applet 标记的语法

下面为 Applet 标记的完整语法:

```
<applet
[archive = archiveList]
code = appletFile. class
width = pixels height = pixels
[codebase = codebaseURL]
[alt = alternateText ]
[name = appletInstanceName ]
[align = alignment ]
[vspace = pixels ] [hspace = pixels ]
>
[ < param name = appletAttribute1 value = value > ]
[ < param name = appletAttribute2 value = value > ]
…
[alternateHTML]
</ applet >
```


5.4.2 Applet 标记描述

1) archive = archiveList——可选属性。描述了一个或多个含有将被“预装”的类和其他资源的 archives。类的装载由带有给定 codebase 的 AppletClassLoader 的一个实例来完成。ArchiveList 中的 archives 以逗号（,）分隔。

2) code = appletFile.class——必选属性。它给定了含有已编译好的 Applet 子类的文件名，扩展名必须为 .class。

3) width = pixels 、height = pixels——必选属性。给出了 Applet 显示区域的初始宽度和高度（以像素为单位），不包括 Applet 所产生的任何窗口或对话框。

4) codebase = codebaseURL——可选属性。为 Applet 文件指定 URL（包含有 Applet 代码的目录）。如果这一属性未指定，则采用文档的 URL。

5）alt = alternateText——可选属性。指定一段可替换的文本，当浏览器能读取 Applet 标记但不能执行 Applet 时，这段文本可作为提示显示出来。

6）name = appletInstanceName——可选属性。为 Applet 实例指定有关名称，从而使得在同一页面上的其他 Applet 能够识别该 Applet 并可互相通信。

7）align = alignment——可选属性。指定了 Applet 的对齐方式。它的可取值与基本的 HTML 中 IMG 标记的相应属性相同，包括：left，right，top，texttop，middle，absmiddle，baseline，bottom 和 absbottom。

8）vspace = pixels hspace = pixels——可选属性。指定了 Applet 与周围文本的垂直间距和水平间距（用像素数表示）。

9）< param name = appletAttribute1 value = value > ——可选属性。这个标记提供了一种可带有由“外部”指定数值的 Applet。Applet 用 getParameter（）方法来存取它们的属性。

5.4.3　HTML

利用符号来描述网页并期待去延长 HTML 文件的符号集叫超文本传输语言。这种语言利用功能能够把页面规则地划分，并能传递给浏览器解释。例如，接下来的在一个 HTML 文件中的 < p > 和 </ p > 之间的文字能以段落的格式显示，但是，以什么样的段落格式展开取决于浏览器。


```
< p >
It began one day in
summer about thirty years ago,
and it happened to four children.
Jane was the oldest
and Mark was the only boy,
and between them they ran everything.
</ p >
```

这是当前所显示的段落（也许你希望能改变字体的大小或者看到格式的变化）：

It began one day in summer about thirty years ago, and it happened to four children. Jane was the oldest and Mark was the only boy, and between them they ran everything.

< p > 和 </ p > 是一种标记符，用来网页划分的描述，标志符通常成双出现，表示文章的开始和结尾。如果标志符像 < something > ，那么与它相匹配的一定是 </ something > 。在超文本标记语言中的“markup”就是 web 文件中“ ⁑ ”中的标记，由浏览器决定哪个功能标记。举例来说，在 < p > 和 </ p > 之间的方式就是段落。

下面有另一个例子：

```
< h3 > This is a level 3 heading </ h3 >
```

它描述了一个标题（主标题下的标题），文本的标题怎么显示取决于浏览器。你的浏览器将显示这样：

This is a level 3 heading

如果一个 HTML 文件只包含 ASCII 码，那么用普通的文本编辑器就能创建（如记事本），没有隐含的显示代码和其他无法打印的字符，如 Word 编辑器常产生的那种（然

而，一个 HTML 文件能够调用网页不只包含 ASCII 码，也包含图片文件和 Applet)。如果一个 html 文件只包含 ASCII 码，那么它在任何一种机器的浏览器上都可方便地实现。

你不必像在例子中显示的那样把标记符以纵向排列，但这样做通常更为方便。接下来的代码也会实现一样的操作：

```
<html> <body>
<h3>Here is an Exciting Applet</h3> <p>
And what an exciting applet it is. With a width of 300 pixels, and a height of
150 pixels,
this applet has it all. </p>
<applet code = "HWloop.class"width = 300 height = 150 > </applet>
</body> </html>
```

这里有一些标记符说明（更多的标记符请在网上或附录中查找)。

<html> 表示 html 文件开始，必须是文件的第一个字符。

</html> 表示 html 文件结束，必须是文件的最后一个字符。

<head> 头文件开始的标志。头文件用来描述或在网页上没有直接可见的一些描述

</head> 头文件结束的标志

<title> 整个网页的一个标题（web 浏览器根据这个能知道它的目的，在页面的头部位置应该可见)

</title> 标题的结束

<body> 文件主体的开始，文件的主体就在屏幕上所见的主要部分。

</body> 文件的主体结束

<hr> 一条水平线，穿过整个屏幕，没有相匹配的标志

这里有个利用这些标志符的网页：

```
……
<html>
<head>
<title>A Boring Title</title>
</head>
……
<body>
<h3>Here is an Exciting Applet</h3>
<p>
And what an exciting applet it is.
With a width of 300 pixels, and a height of 150 pixels,
this applet has it all.
</p>
<applet code = "HWloop.class"width = 300 height = 150>
</applet>
<hr>
</body>
</html>
```

如果忘了写上相对应的标志符，会得到一个非常奇怪的结果。例如，如果忘了与

< h3 > 所匹配的 </ h3 >，那么看起来是在要求由其他文件所组成的一个标题，这看起来是不固定的。

5.5　appletviewer

Applet 通常运行于一个 Web 浏览器中，如 HotJava TM 或 Netscape Navigator，它们有支持 Java 软件程序运行的能力。然而为了简化和加速开发过程，JDK 附带了一个专为查看 Applet 而设计但不支持 HTML 页面查看的工具，这个工具就是 appletviewer。

appletviewer 是不必使用 Web 浏览器即可运行 Applet 的一个 Java 应用程序。用 appletviewer 运行 Applet 时，需要在命令行输入：

```
C: \code> appletviewer Test.html
```

appletviewer 读取命令行中所指定的一个含有 Applet 引用的 HTML 文件。这个 Applet 引用是一个指定了 appletviewer 要装载的代码的 HTML 标记。appletviewer 至少需要以下 HTML 代码：

```
< HTML >
    < APPLET CODE = "Test.class" WIDTH = 200 HEIGHT = 100 >
    </ APPLET >
</ HTML >
```

这个标记的通用格式与任何其他的 HTML 相同，用 < 和 > 两个符号来分隔指令。上例中显示的所有部分都是必需的，同时必须使用 < applet … > 和 </ applet > 。< applet… > 部分指明了代码的入口，以及宽度和高度。

5.6　绘 制 图 形

一个漂亮的用户界面要有生动的颜色、字体，甚至还可以加入合适的图形、图片或者动画，要实现这样的目的，必须熟悉 Graphics 类。

Graphics 类是由 java.awt 包提供的、用于用户低级绘图操作的类，其中包括了许多绘制文字和图形的相关方法，使用 Graphics 类可以绘制线、圆和椭圆、矩形和多边形，显示图像、动画和各种字体。

要实现以上自定义成分，还要借助于 Applet 类的 paint（）方法，由该方法实际地画出图形。当 Applet 运行时，将自动创建一个 Graphics 类的对象，并把这个对象作为参数传递给 paint（）方法，在 paint（）方法中，就可以调用 Graphics 类提供的绘制图形和文字的方法了。paint（）方法形式是：

```
public void paint (Graphics gr)
```

5.6.1　简单的图形绘制

下面的例子是一个利用长方形包围整个图画区域的 applet，然后把另外一个长方形放于中心的位置。

例 5.3

```
import java.applet.Applet;
import java.awt.*;
public class SquareAndRectangle extends Applet
{
  public void paint ( Graphics gr )
  {
    setBackground ( Color.pink );
  //画矩画宽为 100，高为 100 的矩形
    gr.drawRect (10, 10, 100, 100);
//画对角线
    gr.drawLine (10, 10, 110, 100);
                                    //画对角线
    gr.drawLine (110, 10, 10, 110);
  }
}
```

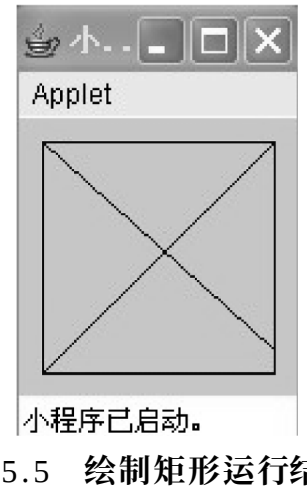


图 5.5　绘制矩形运行结果

当画外面的正方形时，把它的宽度和高度减去一个值后就能得到内部所画区域的边框。编译并运行这个程序，运行结果如图 5.5 所示。

5.6.2　着色

在绘制图形时，经常需要改变一些默认的颜色，用更丰富的颜色输出文字和图形，java.awt 中的 Color 类可以实现这一目的。

利用 setBackground（Color.something）改变纸的颜色，利用 setColor（Color.something）切换不同色的笔，目前为止，都是利用预先定义好的静态类 Color 中的一些静态变量。Color 类提供 13 个静态变量是：white（白色）、lightGray（灰色）、gray（灰色）、darkGray（深灰色）、black（黑色）、red（红色）、pink（粉红色）、orange（橙色）、yellow（黄色）、green（绿色）、magenta（品红色）、cyan（青色）、blue（蓝色）。如下面的语句是把当前颜色设置为蓝色。

```
gr.setColor( Color.blue );
```

5.6.3　绘制直线

利用图形对象中的 drawLine（）方法来绘制直线，方法定义如下：

```
drawLine (int x1, int y1, int x2, int y2)
```

在坐标（x1，y1）和（x2，y2）之间画一条直线，如图 5.6 所示。

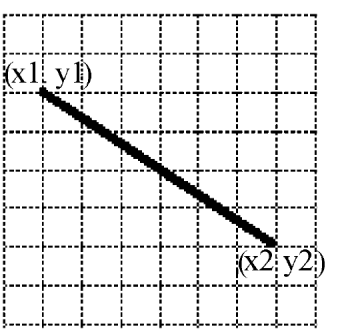


图 5.6　drawLine（）方法参数表示

5.6.4　绘制矩形

利用图形对象中的 drawRect（）方法可以绘制长方形，方法定义如下：

```
drawRect (int x, int y, int width, int height)
```

它利用现有的颜色，画出长方形的轮廓，长方形的左端和右端的值分别是 x 和 x + width，顶端和底端是 y 和 y + height，如图 5.7 所示。当然，也可以用这个方法来画一个正方形。

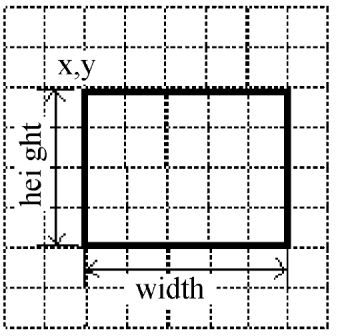


图 5.7　drawRect（）方法参数表示

5.6.5　绘制圆形和椭圆

例 5.4：下面是一个画圆的小程序。

```
import java.applet.Applet;
import java.awt.*;
// 圆的范围是 150, 150
public class JustOneCircle extends Applet
{
  final int radius = 25;
  public void paint ( Graphics gr )
  {
   setBackground ( Color.lightGray );
   gr.drawOval ( (150/2 - radius), (150/2 - radius), radius*2, radius*2 );
   }
}
```


编译并运行这个程序，运行结果如图 5.8 所示。

利用图形对象中的 drawOval（）方法来绘制椭圆，方法定义如下：

```
drawOval ( int x, int y, int width, int height)
```

图 5.8　绘制圆形运行结果

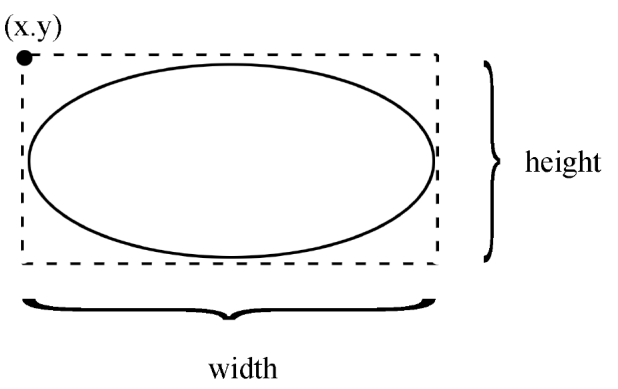


图 5.9　drawOval（）方法参数表示

这里所画的椭圆都恰好放置在由（x，y），width、height 所决定的长方形中。椭圆所在的长方形，它的左上角的坐标就是（x，y）宽和高就是 width 和 height 所表示的值。如图 5.9 所示。drawOval（）方法在长方形内恰好画出椭圆（事实上没有画出长方形）。

圆形是椭圆的特例，Graphics 类没有专门的绘制圆形的方法，当上述两个方法中的矩形宽和高相等时，绘制出来的就是圆形。为了在 Applet 的区域内定位圆心，可以认为圆包含在一个正方形中，那么只要估计在所画区域的正方形的中心就可以了。假如所画的圆宽度和高度都是 150，那么正方形的左上角应该在：

$$x = (150/2 - radius)$$

$$y = (150/2 - radius)$$

可以用相似的计算类定位其他图形的中心，或是你想把图形放在所画区域之外的某些位置。

5.7 Circle 类

有时我们需要一些方便的方法给圆定位。通过定义一个 Circle 类来处理复杂的圆是非常好的。我们有一个重要的设计决定：一个 Circle 对象将成为什么样？如果 Applet 需要画 100 个圆，是否每个圆都要有一个对象？或者是仅仅只有一个 Circle 对象像“画图工具”一样，当我们要进行画圆时，就可以进行调用？

这些问题依赖于我们的目标。首先，我们的目标应该是能很容易地画很多圆，画图将是一个静态的，是一种工作技巧，而不是交互性的。或许“画图工具”对象是最合适的，Circle 对象可定义成像用铅笔和纸来画图一样，在塑料模板上打出各种大小不同的孔。

下面是画 3 个同心圆的 Applet 源程序。

例 5.5

```
import java.applet.Applet;
import java.awt.*;
class Circle
{
  // 变量
   int x, y, radius;
  // 构造器
   public Circle ()
   { x = 0; y = 0; radius = 0; }
   public Circle ( int x, int y, int radius )
   { this.x = x; this.y = y; this.radius = radius; }
  // 方法
   void draw ( Graphics gr )
   {
      int ulX = x - radius ;// 长方形左上角 X 坐标
      int ulY = y - radius ;// 长方形左上角 Y 坐标
      gr.drawOval ( ulX, ulY, 2 * radius, 2 * radius );
   }
}
public class testCircle extends Applet
```

```
{
  Circle circ1 = new Circle ( 100, 100, 10 );
  Circle circ2 = new Circle ( 100, 100, 20 );
  Circle circ3 = new Circle ( 100, 100, 30 );
  public void paint ( Graphics gr )
  {
    circ1.draw ( gr );
    circ2.draw ( gr );
    circ3.draw ( gr );
  }
}
```

编译并运行上面的源程序，运行结果如图 5.10 所示。很显然，利用这个 Circle 对象画同心圆要比直接用 drawOval ( )方法来得简单多了。

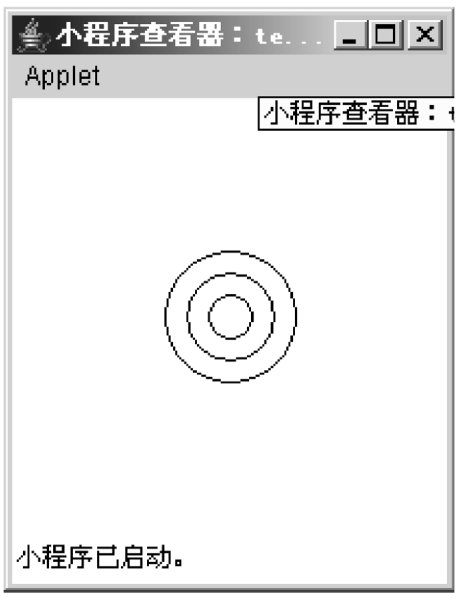


图 5.10　三个同心圆运行结果

```
class Circle
{
  // 变量
  // 构造器
// 方法
}
```


Circle 将是一个画图工具，每次画图工具使用后，在图片上的不仅仅是一些痕迹。

变量

```
int x, y, radius;
```

构造器

```
public Circle ()
{ x = 0; y = 0; radius = 0; }
  public Circle ( int x, int y, int radius )
  {
  ……
  }
```

它将包含左上角的坐标（x，y），同时需要包含圆的长方形的宽和高。但这个 Circle 对象只是用来定位圆的中心和半径是比较好的（不含椭圆）。下面的类定义，可以包含两个构造器：

第一个构造器没有参数，它设定对象变量的默认初始值为 0。通常构造器就是这样被使用的，特别是对象中的数据经常希望被更改的时候。

第二个构造器允许用户详细说明对象每个变量的值，参数名为 x、y 和 radius。

```
public Circle ()
{
  x = 0; y = 0; radius = 0;
}
```

```
public Circle ( int x, int y, int radius )
{
  this.x = x; this.y = y; this.radius = radius;
}
```

保留字 this 用来指出对象的变量，this.x = x 是利用参数“x”来初始化对象的变量“x”。

方法

此外还须有一个画图方法，方法 draw () 将利用 java 语言的方法。

```
drawOval (int x, int Y, int width, int height)
```

方法必须说明在什么区域画一个圆，所以画图方法需设一个专用的参数用来处理图形对象的 Applet。

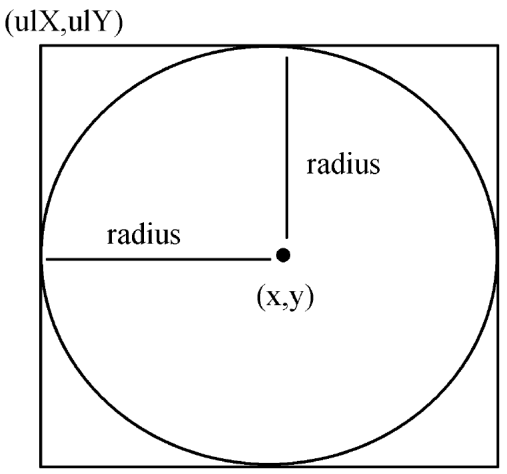


图 5.11 draw ()方法参数表示

```
draw ( Graphics gr )
  {

    int ulX = x - radius;// 长方形左上角的 x 坐标
    int ulY = y - radius;// 长方形左上角的 y 坐标
    gr.drawOval ( ulX, ulY, 2 * radius, 2 * ra-
dius );
  }
```

圆心坐标（x，y）和半径，以及包含圆的正方形的左上角的坐标、宽度和高度的关系如图 5.11 所示。

5.8 利用图形方法画图

本节是一个关于图形的 Applet 例子，这些例子说明规划和图形算法是一样重要的。这一节只有少许的 Java 语言的特性介绍，它所用的大概都是读者已经知道的。

5.8.1 图片的规划

当计划用 Java 去画一个图片时，在方格纸上有个素描是非常有用的，图 5.11 是在方格纸上所展开的图片。

图 5.12 由长方形、线条、椭圆所构成，Java 的图形对象作为一个参数传递给拥有能画出这些方法的 Applet。

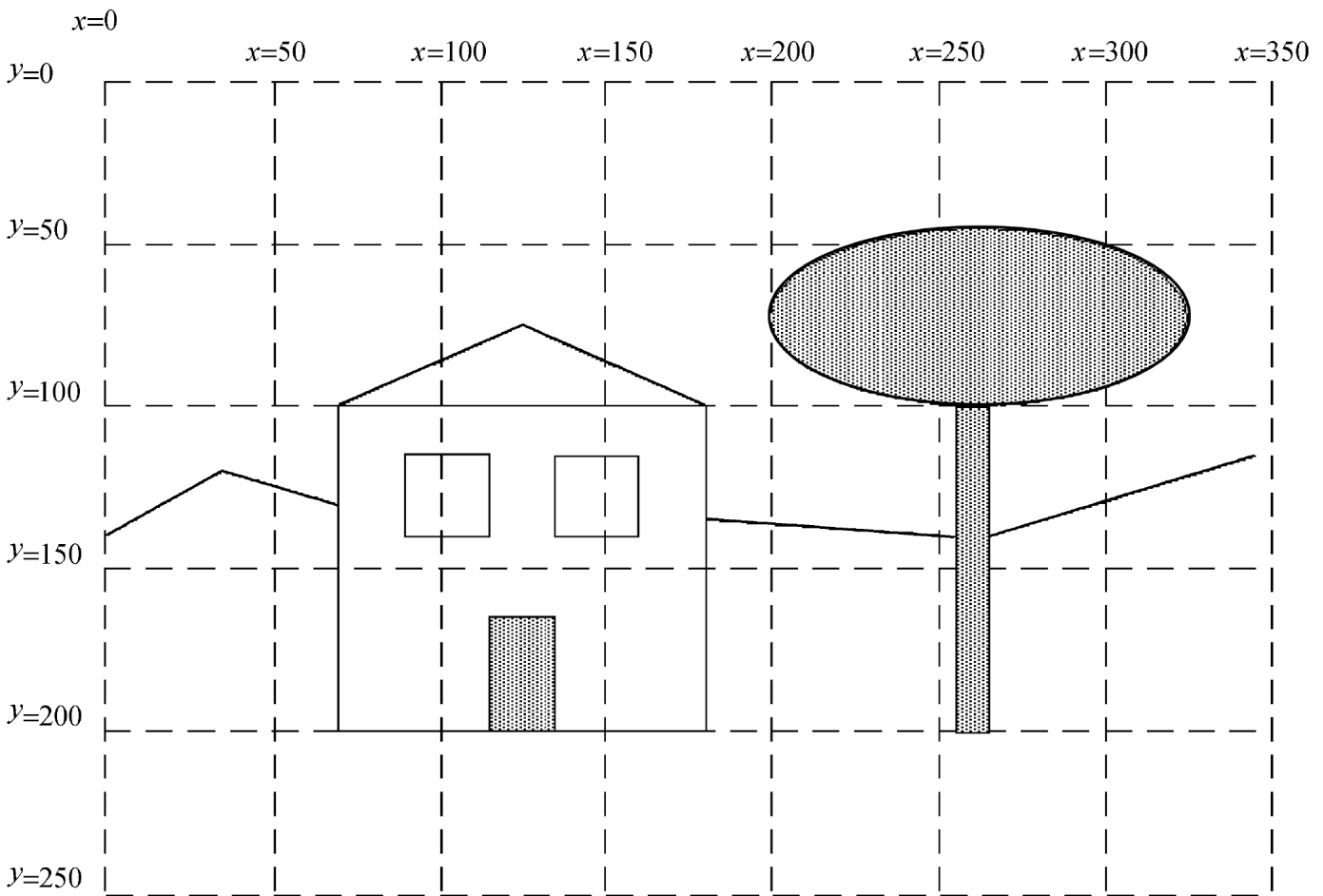


图 5.12　图片的规划

5.8.2　计算坐标

1. 包含所有长方形的 Applet

有了前面的规划后，我们可以通过坐标的计算，先画出所有包含长方形的 Applet。

```
import java.applet.Applet;
import java.awt.*;
// 350*250
public class houseRectangles extends Applet
{
  final int width = 350, height = 250;
  final int houseX = 65, houseY = 100, houseW = 110, houseH = 110 ;
  final int doorX = 120, doorY = 165, doorW = 25, doorH = 40 ;
  final int lWindX = 90, lWindY = 115, lWindW = 30, lWindH = 30 ;
  final int rWindX = 130, rWindY = 115, rWindW = 30, rWindH = 30 ;
  final int trunkX = 255, trunkY = 100, trunkW = 10, trunkH = 100 ;
  public void paint ( Graphics gr )
  {
    gr.setColor (Color.orange );     // there is no Color brown
    gr.drawRect (houseX, houseY, houseW, houseH);// house
    gr.drawRect (doorX, doorY, doorW, doorH );// door
    gr.drawRect (lWindX, lWindY, lWindW, lWindH); // lwind
    gr.drawRect (rWindX, rWindY, rWindW, rWindH); // rwind
```

```
        gr.fillRect (trunkX, trunkY, trunkW, trunkH); // trunk
      }
    }
```

编译并运行这个程序，得到如图 5.13 所示的图片。

因为 fillrect ( ) 方法的使用，所以树干只能是一个固定的长方形，这个方法就像 drawRect ( ) 一样，在轮廓中填入正确的颜色。

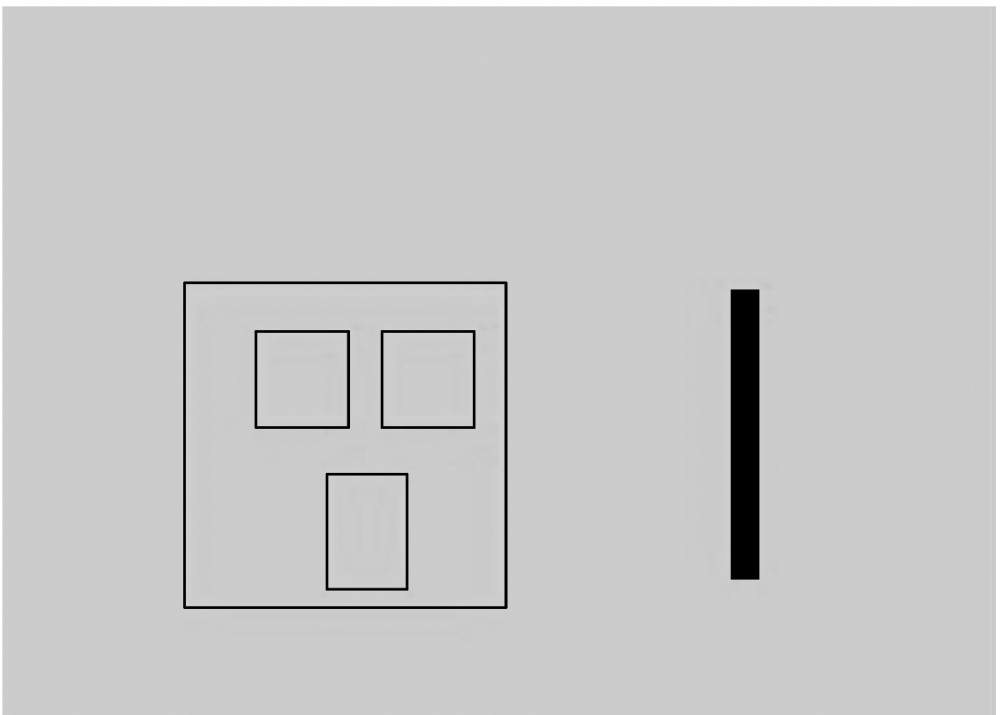


图 5.13 包含所有长方形的 Applet

2. 树叶

图 5.12 中其他同类的点也可以这样计算，例如，窗户可以更为集中，包含树叶的坐标、宽度和高度的计算，下面以树叶为例：

```
final int treeX = 200, treeY = 45, treeW = 125, treeH = 55;   // 树叶
```

为了把树叶加入到树中，接下来的方法调用将加入到 Applet 中：

```
gr.setColor ( Color.green );
gr.fillOval ( treeX, treeY, treeW, treeH );
```

fillOval ( ) 方法和 drawOval ( ) 方法很类似，也是把颜色填入所期望的区域中。

接下来再看一下背景上构成小山的绿色的线，估计线条两端的坐标值 (x，y)，如果恰好有个命名，如 houseX，或是能被运算，那么就画上去。

5.8.3 完整的 Applet

通过规划及坐标运算后，可以编写完整的 Applet 程序了。

```
import java.applet.Applet;
import java.awt.*;

// 图画区域 350 * 250
public class houseComplete extends Applet
```

```
{
  final int width = 350, height = 250;
  final int houseX = 65, houseY = 100, houseW = 110, houseH = 110 ;
  final int doorX = 120, doorY = 165, doorW = 25, doorH = 40 ;
  final int lWindX = 90, lWindY = 115, lWindW = 30, lWindH = 30 ;
  final int rWindX = 130, rWindY = 115, rWindW = 30, rWindH = 30 ;
  final int trunkX = 255, trunkY = 100, trunkW = 10, trunkH = 100 ;
  final int treeX = 200, treeY = 45, treeW = 125, treeH = 55 ;// 树叶

final int L1X1 = 0, L1Y1 = 140, L1X2 = 40, L1Y2 = 115;
  final int L2X1 = L1X2, L2Y1 = L1Y2, L2X2 = houseX, L2Y2 = 130;
  final int L3X1 = houseX + houseW, L3Y1 = 135, L3X2 = trunkX, L3Y2 = 140;
  final int L4X1 = trunkX + trunkW, L4Y1 = L3Y2, L4X2 = width, L4Y2 = 110;

final int roof1X1 = houseX, roof1Y1 = houseY;
  final int roof1X2 = houseX + houswW/2, roof1Y2 = 50;
  final int roof2X1 = roof1X2, roof2Y1 = roof1Y2;
  final int roof2X2 = houseX + houswW, roof2Y2 = houseY;
  final int doorX = houseX + houseW/2 - doorW/2;
  public void paint ( Graphics gr )
  {
    gr.setColor ( Color.orange );// 这里不是棕色
    gr.drawRect ( houseX , houseY , houseW, houseH);// 房子
    gr.fillRect ( doorX , doorY , doorW , doorH );// 门
    gr.drawRect ( lWindX , lWindY , lWindW, lWindH);// 左窗
    gr.drawRect ( rWindX , rWindY , rWindW, rWindH);// 右窗
    gr.fillRect ( trunkX , trunkY , trunkW, trunkH);// 树干

    gr.setColor ( Color.green );
    gr.fillOval ( treeX, treeY, treeW, treeH );

    gr.drawLine ( L1X1, L1Y1, L1X2, L1Y2 );// 线条 1
    gr.drawLine ( L2X1, L2Y1, L2X2, L2Y2 );// 线条 2
    gr.drawLine ( L3X1, L3Y1, L3X2, L3Y2 );// 线条 3
    gr.drawLine ( L4X1, L4Y1, L4X2, L4Y2 );// 线条 4

    gr.setColor ( Color.green );
    gr.drawLine ( roof1X1, roof1Y1, roof1X2, roof1Y2 );
    gr.drawLine ( roof2X1, roof2Y1, roof2X2, roof2Y2 );
  }
}
```


编译并运行上面的程序，运行结果如图 5.14 所示。

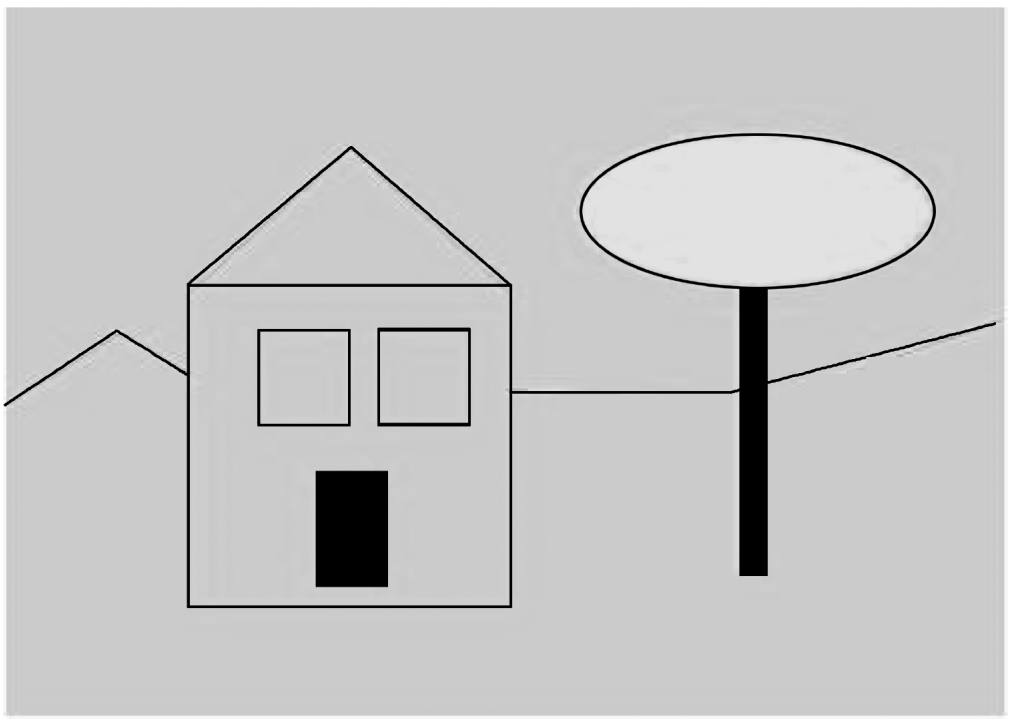


图 5.14 Applet 的运行结果

小 结

Java Applet 是一种特殊的 Java 程序，我们称之为“小应用程序”。Applet 可以被 HTML 页面引用，并可以在支持 Java 的浏览器中执行。

Applet 就是一个扩展了 java.applet.Applet 类的 Java 类。Applet 类提供了使 Applet 能在浏览器上执行的骨干结构，而这个骨干是由 init、start、stop、destroy 这 4 个方法所构成。利用 Applet 类提供这 4 个方法可以构造任意 Applet 框架。

Applet 的开发分为四步：编写 Applet Java 源程序，编译 Applet Java 源程序生成其字节码文件，编写 HTML 文件，调试、运行 Applet 的 HTML 文件 。

学习使用 java.awt 包中的 Graphics 类，绘制简单图形并着色。当 Applet 运行时，浏览器会自动调用它的 paint（）方法，并且创建一个 Graphics 类的对象，对字体、颜色、图形等的控制就是通过这个对象来调用 Graphics 类提供的各种方法而实现的。

习 题

1. 什么浏览器可以用来执行一个 Java Applet ?

2. Java 提供了一个用于用户低级绘图操作的类，这个类是什么？在 Applet 中怎样使用它？

3. 简要列出 Applet 的主要方法。

4. 编写一个 Java Applet 程序，编译并运行这个程序，使之能够在浏览器上显示出“Welcome to java !”。

5. 编写一个用 4 个相同的参数绘制一个椭圆形和一个矩形的 Java Applet。其中椭圆与矩形 4 条边的中点相切。

6. 编写一个程序，该程序绘制 5 个同心圆，圆与圆之间相隔 10 个像素，使用 Graphics 类的 drawOval 方法。

第 6 章　高级的 Java 编程功能

本章要点

递增、递减和其他操作符
短逻辑运算符
再论 for 语句
数组
将数组作为参数
改变数组的方法
对象的数组
线性查找
向量类 Vector

本章学习目标

理解递增、递减和其他操作符并能合理运用
掌握 for 语句的高级用法
理解对象数组、二维数组和多维数组的概念
掌握 Vector 类的用法

6.1 递增、递减和其他操作符

前面已经讲解了算术表达式和赋值语句相关的所有基本知识。本章将讨论其他的操作符，运用这些操作符可以写比较简短的表达式和赋值语句。本节将重点介绍递增操作符（++）、递减操作符（--）、赋值操作符以及调和级数的例子。

6.1.1 递增/ 递减操作符

多数程序中都含有大量的循环，这些循环由一个记录循环次数的变量来控制。很明显，每循环一次，这个变量都会加 1。当然对一个变量加 1 还会在其他许多情况中出现。计算机工程师在观察了许多运行程序之后，发现对一个变量加 1 是使用的最频繁的一个操作。所有处理芯片在处理这个操作上都被设计的非常快。

递增运算符对其运算数加 1，递减运算符对其运算数减 1，因此：

```
x = x + 1;
```

运用递增运算符可以将上面的表达式改写成：

```
x++;
```

同样，语句：

```
x = x - 1;
```

运用递减运算符可以改写成：

```
x--;
```


在前面的例子中，递增或递减运算符采用前缀（prefix）或后缀（postfix）格式都是相同的。但是，当递增或递减运算符作为一个较大的表达式的一部分，就会有重要的不同。如果递增或递减运算符放在其运算数前面，Java 就会在获得该运算数的值之前执行相应的操作，并将其用于表达式的其他部分。如果运算符放在其运算数后面，Java 就会先获得该操作数的值再执行递增或递减运算。例如：

```
x = 42;
y = ++x;
```

这个例子中 y 将被赋值为 43，因为在赋值之前要先执行自加操作。因此 y = ++x 这个语句和下面的两个语句的组合效果是一样的：

```
x = x + 1;
y = x;
```

但是，当我们将程序写成这样时：

```
x = 42;
y = x++;
```

这里在执行 x 的自加之前，先执行了赋值，因此 y 的值应该为 42。当然，在这两个例子中，x 都被赋值为 43。在本例中，程序行 y = x++; 与下面两个语句等价：

```
y = x;
x = x + 1;
```

下面用一个程序来说明 ++ 的用法（-- 的用法完全是一样的）：

```
// Demonstrate ++
```

```
class IncDec {
 public static void main (String args []) {
   int a = 1;
   int b = 2;
   int c;
   int d;
   c = ++b;
   d = a++;
   c++;
   System.out.println ("a = "+ a);
   System.out.println ("b = "+ b);
   System.out.println ("c = "+ c);
   System.out.println ("d = "+ d);
```

该程序的输出如下:

```
a = 2
b = 3
c = 4
d = 1
```

6.1.2　赋值操作符

赋值运算符是一个等号“ = ”。它在 Java 中的运算与在其他计算机语言中的运算一样，其通用格式为:

```
var = expression;
```

对于这样的用法我们在以前的学习中用的也比较多，这里就不再累赘。但读者们可否知道操作符 +，-，*，/ (还有其他符号) 也可以同 = 号用在一起组成复合运算符，比如，下面的代码对 sum 加 5:

```
sum += 5;          // add 5 to sum
```

它和下面的语句有一样的效果:

```
sum = sum + 5;// add 5 to sum
```

下面给出一个复合运算符的列表，如表 6.1 所示。

表 6.1　复合运算符

操作符	操作	例子	结果
=	赋值	sum = 5;	sum = 5
+=	增加并赋值	sum += 5;	sum = sum + 5;
-=	减少并赋值	sum -= 5;	sum = sum - 5;
*=	乘上并赋值	sum *= 5;	sum = sum * 5;
/=	除去并赋值	sum/= 5;	sum = sum/ 5;

这些操作符按照三步来执行，首先，计算等式右边的值，然后，执行和“ = ”号复合的符号运算，最后，将值赋给变量。

6.1.3　调和级数例子

我们的目标是写一个程序计算下面的 sum 值:

$$sum = 1/1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6$$

虽然这个例子看起来像是没有什么意义，不过你将会看到被称作调和级数（harmonic series）的微积分和，程序如下：

```
{
  Public class Harmenic Series
    double value (int limit)
  {
    int term = 1;
    double sum = 0.0;

    while ( term < = limit )
    {
     sum + = 1.0/ term;          // 增加下一个 term 到 sum
     term++ ;                    // 自增 term
    }

    return sum;
  }
}
 class HarmonicTester
{
  public static void main ( String [] args ) throws IOException
  {
    BufferedReader stdin =
     new BufferedReader ( new InputStreamReader (System . in) );
    HarmonicSeries series = new HarmonicSeries ();
    System.out.println ("input the num:");
    int limit = Integer.parseInt (stdin.readLine ());
    System.out.println ("Sum of 6 terms:"+ series.value (limit) );
  }
}
```


关于上述程序的一些需要注意的问题：

1）HarmonicSeries 类描述一个可以求得总和的类。

2）对象的 value（）方法将执行这个计算。

3）虽然没有明显的构造一个 HarmonicSeries 的构造方法，但是会自动生成。

4）HarmonicTester 类建立了 HarmonicSeries 对象，然后使用 value（）方法计算出了总和，并打印出来。

6.2　短逻辑运算符

6.2.1　位逻辑运算符

位逻辑运算符有“与”(AND)、“或”(OR)、“异或 (XOR)”、“非 (NOT)”，分别用“&”、“|”、“^”、“～”表示，表 6.2 显示了每个位逻辑运算符的结果。在继续讨论之前，请记住位运算符应用于每个运算数内的每个单独的位。

表 6.2　位逻辑运算符的结果

A	B	A| B	A & B	A ^ B	～A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

1. 按位非 (NOT)

按位非也叫做补，一元运算符 NOT“～”是对其运算数的每一位取反。例如，数字 42，它的二进制代码为：

```
00101010
```


经过按位非运算成为：

```
11010101
```

2. 按位与 (AND)

按位与运算符“&”，如果两个运算数都是 1，则结果为 1。其他情况下，结果均为 0。看下面的例子：

```
 00101010          42
&00001111          15
 ————————

 00001010          10
```

3. 按位或 (OR)

按位或运算符“|”，任何一个运算数为 1，则结果为 1。如下面的例子所示：

```
 00101010          42
|00001111          15
 ————————

 00101111          47
```

4. 按位异或 (XOR)

按位异或运算符“^”，只有在两个比较的位不同时其结果是 1。否则，结果是 0。下面的例子显示了“^”运算符的效果。这个例子也表明了 XOR 运算符的一个有用的属

性。注意第二个运算数有数字 1 的位，42 对应二进制代码的对应位是如何被转换的。第二个运算数有数字 0 的位，第一个运算数对应位的数字不变。当对某些类型进行位运算时，将会看到这个属性的用处。

```
 00101010          42
^00001111          15
 ————————

 00100101          37
```

5. 位逻辑运算符的应用

下面的例子说明了位逻辑运算符:

```
class BitLogic {
  public static void main (String args []) {
    String binary [] = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };
    int a = 3; // 0 + 2 + 1 或二进制中的 0011
    int b = 6; // 4 + 2 + 0 或二进制中的 0110
    int c = a | b;
    int d = a & b;
      int f = (~a & b) | (a & ~b);
    int g = ~a & 0x0f;

    System.out.println ("a = " + binary [a]);
    System.out.println ("b = " + binary [b]);
    System.out.println ("a|b = " + binary [c]);
    System.out.println ("a&b = " + binary [d]);
    System.out.println ("a^b = " + binary [e]);
    System.out.println ("~a&b|a&~b = " + binary [f]);
    System.out.println ("~a = " + binary [g]);
  }
}
```


在本例中，变量 a 与 b 对应位的组合代表了二进制数所有的 4 种组合模式：00，01，10 和 11。“|”运算符和“&”运算符分别对变量 a 与 b 各个对应位的运算得到了变量 c 和变量 d 的值。对变量 e 和 f 的赋值说明了“^”运算符的功能。字符串数组 binary 代表了 0～15 对应的二进制的值。在本例中，数组各元素的排列顺序显示了变量对应值的二进制代码。数组之所以这样构造是因为变量的值 n 对应的二进制代码可以被正确的存储在数组对应元素 binary [n] 中。例如变量 a 的值为 3，则它的二进制代码对应地存储在数组元素 binary [3] 中。～a 的值与数字 0x0f (对应二进制为 0000 1111) 进行按位与运算的目的是减小～a 的值，保证变量 g 的结果小于 16。因此该程序的运行结果可以用数组 binary 对应的元素来表示，该程序的输出如下:

```
a          = 0011
b          = 0110
a|b        = 0111
a&b        = 0010
a^b        = 0101
~a&b|a&~b  = 0101
~a         = 1100
```

6.2.2　左移位运算符

左移位运算符 < < 使指定值的所有位都左移规定的次数。它的通用格式如下所示：

```
value < < num
```

这里，num 指定要移位值 value 移动的位数。也就是，左移位运算符 < < 使指定值的所有位都左移 num 位。每左移一个位，高阶位都被移出（并且丢弃），并用 0 填充右边。这意味着当左移的运算数是 int 类型时，每移动 1 位，它的第 31 位就要被移出并且丢弃；当左移的运算数是 long 类型时，每移动 1 位，它的第 63 位就要被移出并且丢弃。

在对 byte 和 short 类型的值进行移位运算时必须小心，因为 Java 在对表达式求值时，将自动把这些类型扩大为 int 型，而且，表达式的值也是 int 型 。对 byte 和 short 类型的值进行移位运算的结果是 int 型，而且如果左移不超过 31 位，原来对应各位的值也不会丢弃。但是，如果对一个负的 byte 或者 short 类型的值进行移位运算，它被扩大为 int 型后，它的符号也被扩展。这样，整数值结果的高位就会被 1 填充。因此，为了得到正确的结果，就要舍弃得到结果的高位。这样做的最简单办法是将结果转换为 byte 型。下面的程序说明了这一点：

```
// 按位左移
class ByteShift {
  public static void main (String args []) {
    byte a = 64, b;
    int i;

    i = a < < 2;
    b = (byte) (a < < 2);

    System.out . println ("Original value of a: " + a);
    System.out . println ("i and b: " + i + "" + b);
  }
}
```

该程序产生的输出下所示：

```
Original value of a: 64
i and b: 256 0
```

因变量 a 在赋值表达式中，故被扩大为 int 型，64（01000000）被左移两次生成值 256（100000000）被赋给变量 i。然而，经过左移后，变量 b 中唯一的 1 被移出，低位全部成了 0，因此 b 的值也变成了 0。

既然每次左移都可以使原来的操作数翻倍，程序员们经常使用这个办法来进行快速的 2 的乘法。但是要小心，如果将 1 移进高阶位（31 或 63 位），那么该值将变为负值。下面的程序说明了这一点：

```
// 左移实现快速乘 2
class MultByTwo {
  public static void main (String args []) {
    int i;
    int num = 0xFFFFFFE;

    for (i = 0; i < 4; i + + ) {
      num = num < < 1;
      System . out . println (num);
    }
  }
}
```

该程序的输出如下所示：

```
536870908
1073741816
2147483632
 - 32
```


初值经过仔细选择，以便在左移 4 位后，它会产生 - 32。正如你看到的，当 1 被移进 31 位时，数字被解释为负值。

6.2.3 右移位运算符

右移运算符 > > 使指定值的所有位都右移规定的次数。它的通用格式如下所示：

```
value > > num
```

这里，num 指定要移位值 value 移动的位数。也就是，右移运算符 > > 使指定值的所有位都右移 num 位。

下面的程序片段将值 32 右移 2 次，将结果 8 赋给变量 a：

```
int a = 32;
a = a > > 2; // 结果为 8
```

当值中的某些位被“移出”时，这些位的值将丢弃。例如，下面的程序片段将 35 右移 2 次，它的 2 个低位被移出丢弃，也将结果 8 赋给变量 a：

```
int a = 35;
a = a > > 2; // 结果仍为 8
```

用二进制表示该过程可以更清楚地看到程序的运行过程：

```
00100011  35
 > > 2
00001000  8
```

将值每右移一次，就相当于将该值除以 2 并且舍弃了余数。可以利用这个特点将一个整数进行快速除 2。当然，一定要确保不会将该数原有的任何一位移出。

右移时，被移走的最高位（最左边的位）由原来最高位的数字补充。例如，如果要移走的值为负数，每一次右移都在左边补 1，如果要移走的值为正数，每一次右移都在左边补 0，这叫做符号位扩展（保留符号位），在进行右移操作时用来保持负数的符号。例如，-8 >>1 是 -4，用二进制表示如下：

```
11111000   -8
>>1
11111100   -4
```

一个要注意的有趣问题是，由于符号位扩展（保留符号位）每次都会在高位补 1，因此 -1 右移的结果总是 -1。

有时不希望在右移时保留符号。例如，下面的例子将一个 byte 型的值转换为用十六进制表示。注意右移后的值与 0x0f 进行按位与运算，这样可以舍弃任何的符号位扩展，以便得到的值可以作为定义数组的下标，从而得到对应数组元素代表的十六进制字符。

```
// 舍弃符号位扩展
class HexByte {
  static public void main (String args []) {
    char hex [] = {
      '0','1','2','3','4','5','6','7',
      '8','9','a','b','c','d','e','f'
    };
    byte b = (byte) 0xf1;

    System.out.println ("b = 0x" + hex [ (b >> 4) & 0x0f] + hex [b & 0x0f]);
  }
```


该程序的输出如下：

```
b = 0xf1;
```

6.2.4　位运算符赋值

所有的二进制位运算符都有一种将赋值与位运算组合在一起的简写形式。例如，下面两个语句都是将变量 a 右移 4 位后赋给 a：

```
a = a >> 4;
a >>= 4;
```

同样，下面两个语句都是将表达式 a OR b 运算后的结果赋给 a：

```
a = a | b;
a |= b;
```

下面的程序定义了几个 int 型变量，然后运用位赋值简写的形式将运算后的值赋给相应的变量：

```
class OpBitEquals {
  public static void main (String args []) {
    int a = 1;
    int b = 2;
```

```
        int c = 3;

        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
      }
    }
```

该程序输出如下所示:

```
a = 3
b = 1
c = 6
```

6.3 再论 for 语句

在前面几章曾使用过一个 for 循环的简单格式。通过本节将了解，for 循环是一个功能强大且形式灵活的结构。下面是 for 循环的通用格式:

```
for (初始化表达式; 条件; 控制表式){
  // 循环体
}
```

如果只有一条语句需要重复，大括号就没有必要。

for 循环的执行过程如下。第一步，当循环启动时，先执行其初始化部分。通常，这是设置循环控制变量值的一个表达式，作为控制循环的计数器。重要的是要理解初始化表达式仅被执行一次。下一步，计算条件的值。条件必须是布尔表达式。它通常将循环控制变量与目标值相比较。如果这个表达式为真，则执行循环体；如果为假，则循环终止。再下一步执行循环体的反复部分。这部分通常包含一个用来增加或减少循环控制变量的一个表达式。接下来重复循环，首先计算条件表达式的值，然后执行循环体，接着执行反复表达式。这个过程不断重复直到控制表达式变为假。

下面是使用 for 循环的“tick”程序:

```
// for 循环举例
class ForTick {
  public static void main (String args []) {
    int n;

    for (n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

6.3.1　在 for 循环中声明循环控制变量

控制 for 循环的变量经常只是用于该循环，而不用在程序的其他地方。在这种情况下，可以在循环的初始化部分中声明变量。例如，下面重写了前面的程序，使变量 n 在 for 循环中被声明为整型：

```
// 在 for 循环中声明循环控制变量
class ForTick {
  public static void main (String args []) {
    // n 是循环控制变量
    for (int n = 10; n > 0; n - - )
      System.out.println ("tick " + n);
  }
}
```

在 for 循环内声明变量时，必须记住重要的一点：该变量的作用域在 for 语句执行后就结束了（因此，该变量的作用域就局限于 for 循环内）。在 for 循环外，变量就不存在了。如果在程序的其他地方需要使用循环控制变量，就不能在 for 循环中声明它。

由于循环控制变量不会在程序的其他地方使用，大多数程序员都在 for 循环中来声明它。例如，以下为测试素数的一个简单程序。注意，由于其他地方不需要 i，所以循环控制变量 i 在 for 循环中声明。


```
// 测试是否为素数
class FindPrime {
  public static void main (String args []) {
    int num;
    boolean isPrime = true;

    num = 14;
    for (int i = 2; i < = num/ 2; i++ ) {
      if ( (num % i) = = 0) {
        isPrime = false;
        break;
      }
    }
    if (isPrime) System.out.println ("Prime");
    else System.out.println ("Not Prime");
  }
}
```

6.3.2　逗号的使用方法

有时可能经常需要在初始化和 for 循环的反复部分包含超过一个变量的声明。例如，考虑下面程序的循环部分：

```
class Sample {
  public static void main (String args []) {
```

```
    int a, b;
    b = 4;
    for (a=1; a<b; a++) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      b--;
    }
  }
}
```

如你所看到的，循环被两个相互作用的变量控制。由于循环被两个变量控制，如果两个变量都能被定义在 for 循环中，而变量 b 不需要在 for 循环体外设置将是很方便的。幸好，Java 提供了一个完成此任务的方法。为了允许两个或两个以上的变量控制循环，Java 允许在 for 循环的初始化部分和反复部分声明多个变量，每个变量之间用逗号分开。使用逗号，前面的 for 循环将更高效，改写后的程序如下：

```
class Comma {
  public static void main (String args []) {
    int a, b;

    for (a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```


在本例中，初始化部分把两个变量 a 和 b 都定义了。在循环的反复部分，用两个逗号分开的语句在每次循环重复时都执行。程序输出如下：

```
a = 1
b = 4
a = 2
b = 3
```

注意：C/C++ 中，逗号是一个运算符，能在任何有效的表达式中使用。然而，在 Java 中不是这样。在 Java 中，逗号仅仅是一个分隔符，只适用于 for 循环。

6.3.3 for 循环的一些变化

for 循环支持一些变化，这增加了它的功能和灵活性。for 循环这样灵活是因为它的 3 部分（初始化部分，条件测试部分和反复部分）并不仅用于它们所限定的那些目的。事实上，for 循环的 3 部分能被用于需要的任何目的。让我们看一些例子。

最普通的变化之一包含在条件表达式中。具体地说，条件表达式可以不需要用循环变量和目标值的比较来测试循环条件。事实上，控制 for 循环的条件可以是任何布尔表达式。例如，考虑下列程序片段：

```
boolean done = false;
for (int i=1; !done; i++) {
```

```
    // ...
     if (interrupted ()) done = true;
   }
```

在本例中，for 循环将一直运行，直到布尔型变量 done 被设置为真。for 循环的条件部分不测试值 i。

下面是 for 循环的另外一个有趣的变化。在 Java 中可以使 for 循环的初始化、条件或者反复部分中的任何或者全部都为空，如下面的程序：

```
// 在 for 语句中可以部分为空
class ForVar {
  public static void main (String args []) {
    int i;
    boolean done = false;

    i = 0;
    for ( ; !done; ) {
      System.out.println ("i is " + i);
      if (i = = 10) done = true;
      i ++ ;
    }
  }
}
```


本例中，初始化部分和反复部分被移到了 for 循环以外。这样，for 循环的初始化部分和反复部分是空的。在这个简单的例子中，for 循环中没有值，确实，这种风格被认为是相当差的，有时这种风格也是有用的。例如，如果初始条件由程序中其他部分的复杂表达式来定义，或者循环控制变量的改变由发生在循环体内的行为决定，而且这种改变是一种非顺序的方式，这种情况下，可以使 for 循环的这些部分为空。

下面是 for 循环变化的又一种方式。如果 for 循环的 3 个部分全为空，就可以创建一个无限循环（从来不停止的循环）。例如：

```
for ( ; ; ) {
  // ...
}
```

这个循环将始终运行，因为没有使它终止的条件。尽管有一些程序，例如操作系统命令处理器，需要无限循环，但大多数“无限循环”实际上是具有特殊终止要求的循环。不久将看到如何不用正常的条件表达式来终止这种类型的循环。

6.3.4　循环嵌套

和其他编程语言一样，Java 允许循环嵌套。也就是，一个循环在另一个循环之内。例如，下面的程序就是循环嵌套：

```
// 循环嵌套
class Nested {
  public static void main (String args []) {
    int i, j;
```

```
        for (i = 0; i < 10; i++) {
          for (j = i; j < 10; j++)
            System.out.print (".");
          System.out.println ();
        }
      }
    }
```

该程序的输出如下所示：

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . .
. . . .
. . .
. .
.
```


6.4 数组

数组是相同类型变量的集合，可以使用共同的名字引用它。数组可被定义为任何类型，可以是一维或多维。数组中的一个特别要素是通过下标来访问它。数组提供了一种将有联系的信息分组的便利方法。

请注意，Java 数组的工作原理与 C/C++ 不同。

6.4.1 一维数组

一维数组实质上是相同类型变量列表。要创建一个数组，必须首先定义数组变量所需的类型。通用的一维数组的声明格式是：

```
type [] var;
```

其中，type 定义了数组的基本类型。基本类型决定了组成数组的每一个基本元素的数据类型。这样，数组的基本类型决定了数组存储的数据类型。例如，下面的例子定义了数据类型为 int，名为 month_days 的数组。

```
int month_days [];
```

尽管该例子定义了 month_days 是一个数组变量，但实际上没有数组存储空间的存在。month_days 的值被设置为空，它代表一个数组没有值。为了使数组 month_days 成为实际的、物理上存在的整型数组，必须用运算符 new 来为其分配地址并且把它赋给 month_days。运算符 new 是专门用来分配内存的运算符。

你将在后面章节中更进一步了解运算符 new，但是现在需要使用它来为数组分配内存。当运算符 new 被应用到一维数组时，它的一般形式如下：

```
array_var = new type [size];
```

其中，type 指定被分配的数据类型，size 指定数组中变量的个数，array _ var 是被链接到数组的数组变量。也就是，使用运算符 new 来分配数组，必须指定数组元素的类型和数组元素的个数。用运算符 new 分配数组后，数组中的元素将会被自动初始化为零。下面的例子分配了一个 12 个整型元素的数组并把它们和数组 month _ days 链接起来。

```
month_days = new int [12];
```

通过这个语句的执行，数组 month _ days 将会指向 12 个整数，而且数组中的所有元素将被初始化为零。

让我们回顾一下上面的过程：获得一个数组需要 2 步。第一步，必须定义变量所需的类型。第二步，必须使用运算符 new 来为数组所要存储的数据分配内存，并把它们分配给数组变量。这样 Java 中的数组被动态地分配。如果动态分配的概念对你陌生，别担心，它将在本书的后面详细讨论。

一旦分配了一个数组，可以在方括号内指定它的下标来访问数组中特定的元素。所有的数组下标从零开始。例如，下面的语句将值 28 赋给数组 month _ days 的第二个元素。

```
month_days [1] = 28;
```

下面的程序显示存储在下标为 3 数组元素的值：

```
System.out.println ( month_days [ 3 ]);
```

综上所述，下面程序定义的数组存储了每月的天数。


```
// 一维数组举例
class Array {
  public static void main (String args []) {
    int month_days [];
    month_days = new int [12];
    month_days [0] = 31;
    month_days [1] = 28;
    month_days [2] = 31;
    month_days [3] = 30;
    month_days [4] = 31;
    month_days [5] = 30;
    month_days [6] = 31;
    month_days [7] = 31;
    month_days [8] = 30;
    month_days [9] = 31;
    month_days [10] = 30;
    month_days [11] = 31;
    System.out.println ("April has " + month_days [3] + " days.");
  }
}
```

运行这个程序时，它打印出 4 月份的天数。如前面提到的，Java 数组下标从零开始，因此 4 月份的天数数组元素为 month _ days [3]。

当然可以将对数组变量的声明和对数组本身的分配一起进行，如下所示：

```
int month_days [] = new int [12];
```

这将是通常编写 Java 程序的专业做法。

数组可以在声明时被初始化。这个过程和简单类型初始化的过程一样。数组的初始化就是包括在花括号之内用逗号分开的表达式的列表。逗号分开了数组元素的值。Java 会自动地分配一个足够大的空间来保存指定的初始化元素的个数，而不必使用运算符 new。例如，为了存储每月中的天数，下面的程序定义了一个初始化的整数数组：

```
// 上述程序改进版本
class AutoAray {
  public static void main (String args []) {
    int month_dadys [] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    Sytem . cut . println ("April has "+ month_days [3] + "days .");
  }
}
```

运行这个程序时，你会看到它和前一个程序产生的输出一样。

Java 严格地检查以保证不会意外地去存储或引用在数组范围以外的值。Java 的运行系统会检查以确保所有的数组下标都在正确的范围以内（在这方面，Java 与 C/ C++ 从根本上不同，C/ C++ 不提供运行边界检查）。例如，运行系统将检查数组 month _ days 的每个下标的值以保证它包括在 0～11 之间。如果企图访问数组边界以外（负数或比数组边界大）的元素，将引起运行错误。

下面的例子运用一维数组来计算一组数字的平均数。

```
// 一维数组的平均值
class Average {
  public static void main (String args []) {
    double nums [] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;

    for (i=0; i<5; i++)
      result = result + nums [i];

    System . out . println ("Average is " + result/ 5);
  }
}
```

6.4.2 二维数组和多维数组

在 Java 中，多维数组实际上是数组的数组。定义多维数组变量要将每个维数放在它们各自的方括号中。例如，下面语句定义了一个名为 twoD 的二维数组变量。

```
int twoD [] [] = new int [4] [5];
```

该语句分配了一个 4 行 5 列的数组并把它分配给数组 twoD。实际上这个矩阵表示了 int 类型的数组的数组被实现的过程。概念上，这个数组可以用图 6.1 来表示。

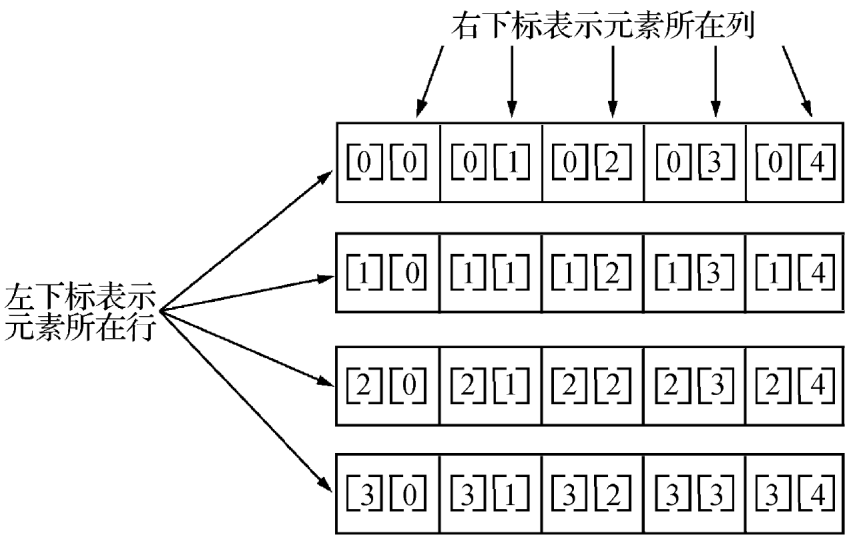


图 6.1　二维数组（4 行 5 列）的概念性表示

```
// 二维数组举例
class TwoDArray {
  public static void main (String args []) {
    int twoD [] [] = new int [4] [5];
    int i, j, k = 0;

    for (i = 0; i < 4; i++)
      for (j = 0; j < 5; j++) {
        twoD [i] [j] = k;
        k++;
      }

    for (i = 0; i < 4; i++) {
      for (j = 0; j < 5; j++)
        System.out.print (twoD [i] [j] + "");
      System.out.println ();
    }
  }
}
```


程序运行的结果如下：

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

给多维数组分配内存时，只需指定第一个（最左边）维数的内存即可。可以单独地给余下的维数分配内存。例如，下面的程序在数组 twoD 被定义时给它的第一个维数分配内存，对第二维则是手工分配地址。

```
int twoD [] [] = new int [4] [];
twoD [0] = new int [5];
twoD [1] = new int [5];
```

```
twoD [2] = new int [5];
twoD [3] = new int [5];
```

尽管在这种情形下单独地给第二维分配内存没有什么优点，但在其他情形下就不同了。例如，当手工分配内存时，不需要给每个维数相同数量的元素分配内存。如前面所说，既然多维数组实际上是数组的数组，每个数组的维数在你的控制之下。例如，下列程序定义了一个二维数组，它的第二维的大小是不相等的。

```
// 自定义不同大小的二维数组
class TwoDAgain {
  public static void main (String args []) {
    int twoD [] [] = new int [4] [];
    twoD [0] = new int [1];
    twoD [1] = new int [2];
    twoD [2] = new int [3];
    twoD [3] = new int [4];
    int i, j, k = 0;

    for (i = 0; i < 4; i++)
      for (j = 0; j < i + 1; j++) {
        twoD [i] [j] = k;
        k++;
      }

    for (i = 0; i < 4; i++) {
      for (j = 0; j < i + 1; j++)
        System.out.print (twoD [i] [j] + " ");
      System.out.println ();
    }
  }
}
```


该程序产生的输出如下：

```
0
1 2
3 4 5
6 7 8 9
```

该程序定义的数组如图 6.2 所示。

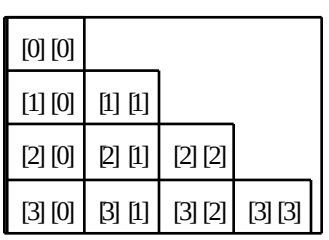


图 6.2 程序定义的数组

对于大多数应用程序，我们不推荐使用不规则多维数组，因为它们的运行与人们期望的相反。但是，不规则多维数组在某些情况下使用效率较高。例如，如果需要一个很大的二维数组，而它仅仅被稀疏地占用（即其中一维的元素不是全被使用），这时不规则数组可能是一个完美的解决方案。

初始化多维数组是可能的。初始化多维数组只不过是把每一维的初始化列表用它自己的大括号括起来即可。下面的程序产生一个矩阵，该矩阵的元素包括数组下标的行和列的积。同时注意在数组的初始化中可以像用常量一样用表达式。

```
// 初始化二维数组
class Matrix {
  public static void main (String args []) {
    double m [] [] = {
       { 0 * 0, 1 * 0, 2 * 0, 3 * 0 },
       { 0 * 1, 1 * 1, 2 * 1, 3 * 1 },
       { 0 * 2, 1 * 2, 2 * 2, 3 * 2 },
       { 0 * 3, 1 * 3, 2 * 3, 3 * 3 }
    };
    int i, j;

    for (i = 0; i < 4; i ++ ) {
      for (j = 0; j < 4; j ++ )
        System.out.print (m [i] [j] + " ");
      System.out.println ();
    }
  }
}
```


运行这个程序时，将得到下面的输出：

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

数组中的每一行就像初始化表指定的那样被初始化。

让我们再看一个使用多维数组的例子。下面的程序首先产生一个 3×4×5 的三维数组，然后装入用它的下标之积生成的每个元素，最后显示了该数组。

```
// 二维数组举例
class threeDMatrix {
  public static void main (String args []) {
    int threeD [] [] [] = new int [3] [4] [5];
    int i, j, k;

    for (i = 0; i < 3; i ++ )
      for (j = 0; j < 4; j ++ )
        for (k = 0; k < 5; k ++ )
          threeD [i] [j] [k] = i * j * k;

    for (i = 0; i < 3; i ++ ) {
      for (j = 0; j < 4; j ++ ) {
```

```
            for (k = 0; k < 5; k ++)
          System.out.print (threeD [i] [j] [k] + "");
          system.out.println ();
        }
          System.out.println ();
      }
    }
  }
```

该程序的输出如下:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```


6.4.3 另一种数组声明语法

声明数组还有第二种格式:

```
type [ ] var - name;
```

这里，方括号紧跟在类型标识符 type 的后面，而不是跟在数组变量名的后面。例如，下面的两个定义是等价的:

```
int al [] = new int [3];
int [] a2 = new int [3];
```

下面的两个定义也是等价的:

```
char twod1 [] [] = new char [3] [4];
char [] [] twod2 = new char [3] [4];
```

包含这种数组声明格式主要是为了方便。

6.4.4 将数组作为参数

通常，数组（和其他的数据形式）都是作为参数传给指定的方法。每个方法都像是一个与数组相关的工具。比如，可能存在一个 findMaximum（查最大值）方法或一个 sumElements（求元素总和）的方法。

```
import java.io. * ;
```

```
class ArrayOps
{
  void print ( int [] x )
  {
    for ( int index = 0; index < x . length; index + + )
      System . out . print ( x [index] + "");
    System . out . println ();
  }
}

class ArrayDemo
{
```

当方法调用 operate. print （ar2）时实参 ar2 连到了这个方法的参数 x，如图 6.3 所示。

通过将参数和方法连在一起，可以将相同的方法和不同的数据放到一起使用。因此这个方法就成为一件可以在很多种情况下使用的有用的工具。

注意，图 6.3 中，连接 ar2 的箭头和连接 x 的箭头是不同的，虽然它们指向了相同的对象。当这个程序运行时，在计算机内存中，x 的位模式和 ar2 的位模式是相同的，都包含了指向这个数组对象的单一位模式。

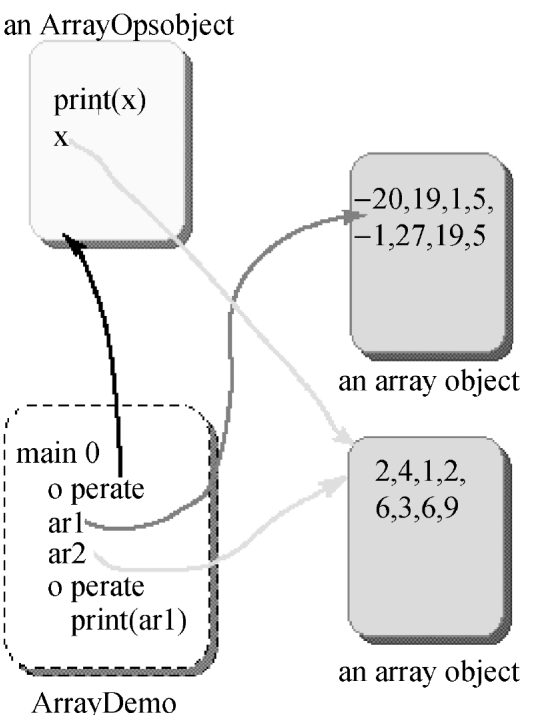


图 6.3　将数组作为参数的举例

6.5　对象的数组

一个数组中的全部元素必须是同一类型的。到目前为止，元素已经有 int、double 和 char 的初始类型。然而，数组元素可以是任何数据类型，包括对象引用。本节讨论对象引用的数组。

6.5.1　对象引用数组

一个对象仅在它被构造后才存在，它一旦存在，一个对象能通过一个引用被访问，该引用往往被保存在一个引用变量例如 str 中。例如，接下来的第一个声明是声明一个引用变量，然后构造一个对象并把它的一个引用放在 str 中：

```
String str;                    / / 声明一个引用变量赋初始值
str = "Hello World";
```

当然，声明一个引用变量而不对其引用（有时是需要的）也是可以的。在不同时间对不同对象用相同的引用变量也是可以的。

```
String str;
str = "Hello World";
str = "Good - by";
```

6.5.2 字符串引用数组

这里声明了一个引用变量 strArray，它将作为一个数组对象。它的每一项或许是一个字符串对象的一个引用（但是目前还没有分配数组的存储空间）。

```
String [] strArray;
```

生成一个 8 个字符串引用的数组：

```
strArray = new String [8];
```

现在 strArray 作为一个数组对象。该数组对象有 8 项（元素）然而这些项中还没有对象存在，对象引用数组的这些项被自动初始化为 null，这个特殊值表示“没有对象”。

现在，实际生成一个字符串并把它的引用保存在数组的第 0 项，如下所示：

```
strArray [0] = "Hello";
```

记住用字符串对象时可以不用 new（它自动产生）来分配。

6.5.3 一个例子

下面我们用一个例子来演示对象数组的用法：

```
class StringArray
{
  public static void main( String [] args )
  {
    String [] strArray = new String [8];
    strArray [0] = "Hello";
    strArray [1] = "World";
    strArray [2] = "Greetings";
    strArray [3] = "Jupiter";
    strArray [ strArray . length - 1 ] = "the end";
    for (int j = 0; j < strArray . length; j ++ )
      if ( strArray [j] ! = null )
        System . out . println ( "Slot " + j + ": " + strArray [j] );
      else
        System . out . println ( "Slot " + j + ": " + "empty");
  }
}
```


第 1 个运行这段代码的程序员将看到：

```
Slot 0: Hello
Slot 1: World
Slot 2: Greetings
Slot 3: Jupiter
Slot 4: empty
Slot 5: empty
Slot 6: empty
```

```
Slot 7: the end
```

6.6　线 性 查 找

本节构造一个包括线性查找方法的 Searcher 类，制造一个静态的线性查找方法。这将使我们在不构造一个 Searcher 对象的情况下使用该方法，下面是程序的框架：

```
class Searcher
{
  // 在字符串数组中检索目标，如果找到则返回下标，否则返回 - 1
   public static int search (_____array, _____ target )
   {
   ......// 实现线性检索
   }
}

class SearchTester
{
   public static void main ( String [] args )
   {
      final int theSize = 20 ;
      String [] strArray = new String [ theSize ] ;

      . . . . . // 将值置入数组 strArray 中

     // 调用静态检索方法
      int where = Searcher.search ( strArray, "小王");
      if ( where > = 0 )
        System.out.println ("Target found in slot " + where );
      else
        System.out.println ("Target not found");

   }
}
```


实参 array 的类型 String [] 说明字符串的一个数组被期望，但没说明这个数组有多长，也要注意在 main () 中静态方法怎样被使用。继续程序，for - loop 将一个个地从 0 项开始检查数组的每一项。

```
class Searcher
{
  // 在字符串数组中检索目标，如果检索到则返回下标，否则返回 - 1
   public static int search ( String [] array, String target )
   {
      for ( int j = 0; j _____ array.length; j++ )
```

```
        if ( array [j] _____ null )
    }
  }

  class SearchTester
  {
    public static void main ( String [] args )
    {
      . . . . . .
      int where = Searcher.search ( strArray, "Peoria");
      . . . . . .
    }
  }
```

然而，如果它没满，不是数组的所有项都将包含一个字符串引用，包含 null 的项必须被跳过。if 语句仅允许非空项使用，并测试数组的 j 项是否是与目标匹配的字符串，换句话说，我们想测试两个字符串的内容是否一致。

```
class Searcher
{
  // 在字符串数组中检索目标，如果检索到则返回下标，否则返回 - 1
  public static int search ( String [] array, String target )
  {
     for ( int j = 0; j < array.length; j ++ )
       if ( array [j] ! = null )
         // do something here with a non - null slot


  }
}
class SearchTester
{
  public static void main ( String [] args )
  {
    . . . . . .
    int where = Searcher.search ( strArray, "小王");
    . . . . . .
  }
}
```


下面的程序有更多的查找方法，看看第二个 if 语句，它便于找到目标后就马上以目标被找到的项号返回给调用者。

```
class Searcher
{
  // 在字符串数组中检索目标，如果检索到则返回下标，否则返回 - 1
  public static int search ( String [] array, String target )
```

```
  {
     for ( int j = 0; j < array.length; j ++ )
       if ( array [j] != null )
         if ( array [j] .equals ( target ) ) return _________;
     return _________ ;// 没有检索到相匹配的字符串，返回
  }
}

class SearchTester
{
  public static void main ( String [] args )
  {
     . . . . . .
     int where = Searcher.search ( strArray, "小王");
     . . . . . .
  }
}
```

第二个 return 语句不是循环体的一部分，它被执行仅当数组的每项都被检查而没有找到目标时。

这里有一个完整的程序。for...loop 一旦找到包含目标的项，该方法就把该项的项号返回给调用者，如果 for...loop 到达数组结尾，就返回 - 1。


```
class Searcher
{
  //在字符串数组中检索目标，如果检索到则返回下标，否则返回 - 1
  public static int search ( String [] array, String target )
  {
     for ( int j = 0; j < array.length; j ++ )
       if ( array [j] != null )
         if ( array [j] .equals ( target ) ) return j ;// 检索到目标

     return - 1 ;// 没有检索到目标
  }
}

class SearchTester
{
  public static void main ( String [] args )
  {
     final int theSize = 20 ;

     String [] strArray = new String [ theSize ] ;
    strArray [0] = "小明";
```

```
        strArray [1] = "小李";
        strArray [2] = "小张";
        strArray [3] = "小莫";
        strArray [4] = "小王";
        strArray [6] = "小林";
        strArray [7] = "小赵";
        strArray [8] = "小孙";

        for (int j = 0; j < strArray.length; j++ )
          if ( strArray [j] != null )
            System.out.println ( j + ": " + strArray [j] );
        // 检索"小王"
        int where = Searcher.search ( strArray, "小王");
        if ( where >= 0 )
          System.out.println ("Target found in slot " + where );
        else
          System.out.println ("Target not found");

    }
}
```


下面添加一个应用，查找一个名字并打印那个人的电话号码，名字和电话号码表被作为 PhoneEntry 对象表实现。

```
class PhoneEntry
{
  String name;      // 人名
  String phone;     // 电话号码

  // constructor
  PhoneEntry ( String n, String p )
  {
    name = n; phone = p;
  }
}
```

电话号码是一个字符串（不是整型）因为它不需做算术，还因为它可能包括冒号和破折号，main ( ) 方法有这个数组：

```
PhoneEntry [] phoneBook = new PhoneEntry [ 5 ] ;
```

这里有一个应用的框架：

```
class PhoneEntry
{
  String name;      // 人名
  String phone;     // 电话号码
```

```
  PhoneEntry ( String n, String p )
  {
    name = n; phone = p;
  }
}

class PhoneBook
{
  PhoneEntry [] phoneBook;

  PhoneBook ()     // 构造函数
  {
    phoneBook = new PhoneEntry [ 5 ] ;

    // 导入电话本数据
    . . . .
  }

  PhoneEntry search ( String targetName )
  {
    // 用线性方法实现人名的检索
    . . . .
  }
}

class PhoneBookTester
{
  public static void main ( String [] args )
  {
    PhoneBook pb = new PhoneBook ();

    // 检索"小周"
    PhoneEntry entry =
        pb.search ( "小周");

    if ( entry ! = null )
      System.out.println ( entry.name +
          ": " + entry.phone );
    else
      System.out.println ("Name not found");

  }
}
```

PhoneBook 类包含数据和一个查找方法，它的查找方法将返回一个匹配被查找名字的 PhoneEntry 的引用。

6.6.1 构造函数

电话簿的数据通常来自一个磁盘文件，在例程中，它将是构造函数中的“hard wired”。

```
class PhoneEntry
{
  String name; // 人名
  String phone; // 电话号码

  PhoneEntry ( String n, String p )
  {
    name = n; phone = p;
  }
}

class PhoneBook
{
  PhoneEntry [] phoneBook;

  PhoneBook ()     // 构造函数
  {
    phoneBook = new PhoneEntry [ 5 ] ;

    phoneBook [0] = new PhoneEntry ( "小王", "0571 - 8320107");
    phoneBook [1] = new PhoneEntry ( "小周", "0571 - 8320108");
    phoneBook [2] = new PhoneEntry ( "小林", "0571 - 8320777");
    phoneBook [3] = new PhoneEntry ( "小李", "0571 - 8320857");
    phoneBook [4] = new PhoneEntry ( "小莫", "0571 - 8320897");
  }

  PhoneEntry search ( String targetName )
  {
    . . . .    // 用线性方法实现人名检索
  }
}
```


假定数组的每一项装有一个 PhoneEntry 对象的引用。

6.6.2 线性查找

如果线性查找不必检验每个数组空项，那么它将被简化，这里有一个完成一部分的 search ( ) 的一些代码。

```
class PhoneEntry
{
  String name; // 人名
  String phone; // 电话号码

  . . . . .
}

class PhoneBook
{

    PhoneEntry [] phoneBook;

    PhoneBook () // 构造函数
    {
      phoneBook = new PhoneEntry [ 5 ] ;

      . . . . . .
  }
```



```
    PhoneEntry search ( String targetName )
    {
      // 用线性方法实现人名的检索

      for ( int j = 0; j < phoneBook. _____ ; j ++ )
      {
        if ( phoneBook [ j ] .name.equals ( _____ ) )
          return phoneBook [ j ];
      }

      return null;
    }
  }
```

回忆起 search () 返回正确项的一个引用，或者如果该项没找到就返回空。

6.6.3　完整的查找

这里有一个完整的 search () 方法。

```
    PhoneEntry search ( String targetName )
    {
      for ( int j = 0; j < phoneBook.length; j ++ )
      {
        if ( phoneBook [ j ] .name.equals ( targetName ) )
```

```
            return phoneBook [ j ];
        }

        return null;
    }
```

你或许对这个表达式感到不舒服:

```
phoneBook [ j ] .name.equals ( targetName )
```

一条条地看一下:

phoneBook [j] 包含一个对一个 PhoneEntry 对象的引用。

PhoneEntry 对象包含变量 name 的一个实例。

name 是一个对字符串对象的一个引用。

所有字符串对象所拥有的方法中，其中一个是 equals ( ) 方法。

以 targetName 当作另一个字符串来用这个 equals ( ) 方法。

整个表达式是求 true 或 false。

这里有一个完整的程序。

```
class PhoneEntry
{
  String name;      // 人名
  String phone;     // 电话号码
  PhoneEntry ( String n, String p)
  {
    name = n; phone = p;
  }
}

class PhoneBook
{
 PhoneEntry [] phoneBook;
 PhoneBook ()// 构造函数
 {
   phoneBook = new PhoneEntry [ 5 ] ;
   phoneBook [0] = new PhoneEntry
     ("小王", "0571 - 8320107");
   phoneBook [1] = new PhoneEntry
   ("小周", "0571 - 8320108");
   phoneBook [2] = new PhoneEntry
     ("小林", "0571 - 8320777");
   phoneBook [3] = new PhoneEntry
     ("小李", "0571 - 8320857");
   phoneBook [4] = new PhoneEntry
     ("小莫", "0571 - 8320897");
 }
```

```
 PhoneEntry search ( String targetName )
 {
   for (int j = 0; j < phoneBook.length; j ++ )
   {
     if ( phoneBook [ j ] .
       name.equals ( targetName))
         return phoneBook [ j ];
    }
    return null;
 }
}

class PhoneBookTester
{
 public static void main (String [] args)
 {
   PhoneBook pb = new PhoneBook ();

  // 检索"小李"
   PhoneEntry entry =
    pb.search ( "小李");
   if ( entry ! = null )
    System.out.println ( entry.name +
      ": " + entry.phone );
   else
     System.out.println ("Name not found");

 }
}
```


6.7　向量类 Vector

6.7.1　向量类 Vector 的引入

对于程序，操作列在表单里的数据是很普通的，你已经知道怎样使用指针，指针是 Java 和大部分程序语言的基本特征。表是很常用的，Java 开发工具包里包括 Vector 类，这种类像指针一样使用，而且有附加的方法和特征。

同于指针的地方，Vector 包括使用整型的基础，然而，不同于指针，Vector 的大小扩展了是否需要把项目附加上去。

一旦一个指针对象被构造，位的数量就被改变了。如果一个指针对象被用作一个

完整的程序运行，它的长度必须足够长以适应所有预期的数据。其实预先是很难猜出所需要的长度的。如果无论预先的指针有多长，在增加数据的地方都能有一个很好的指针，那将是最好的了。

在例子中，基本变量 list 被重新构造成对象。旧的信息没有被自动转移。为了保存旧信息，程序将不得不在它被构造以后复制它到一个新的指针。这并不是很难，但是相当麻烦。

在程序中，斜线被重设为 null，这个值是因为它没有引用任何对象。

Vector 类是构建在指针的基础之上的。Vector 对象包括对象的指针，这些对象涉及到管理指针的方法。Vector 的最大的方便是可以把元素增加上去无论它预先有多大。Vector 的大小将被自动的增加而且信息也不会丢失。

然而，这种方便是有代价的:

1) Vector 的元素是 object references，而不是像 int 和 double 那样的原始数据了。

2) 使用 Vector 比直接使用指针降低了效率。

3) Vector 的元素是 Object 的参考。这就意味着当使用来自 Object 的数据时将一直需要这种类型。

6.7.2 Vector 类的基础

为 Object 类声明一个自动变量，可以这样做:

```
Vector myVector;
```

不用写出预先想存储的类的类型，Vector 就像一个 Object 参考的指针一样。这就意味着任何对象的参考都可以存储在 Vector 中。为一个预先未指明的容量声明一个变量和构造一个指针可以这样做:

```
Vector myVector = new Vector ();
```

这不是最有效的。如果知道需要多少容量，就把 Vector 指向它。为一个预先设定容量为 15 的 Vector 类声明一个变量和构造一个指针:

```
Vector myVector = new Vector (15);
```

这个最初设定的容量大小是 Vector 指定的。如果增加更多的元素可以把它扩展到超出它原有的大小。扩展 Vector 的容量是很慢的，而且如果经常这样做会降低程序的运行速度。为了更好的控制，要指定出多少个新的位在每次 Vector 扩展的时候将被增加:

```
Vector myVector = new Vector (15, 5);
```

现在当需要附加的容量的时候，一次可以增加 5 个位。不需要把新的位都填满，但是当需要的时候可以使用它们。

一个指针对象有容量和大小，用 capacity () 方法找出当前指针的容量。用 size () 方法找出当前指针的大小。

capacity 是可利用的位的个数。

size 是有数据的位的个数。

从 0 位开始到 N - 1 位来存放数据。

可以跳过某些位从 N 位提取数据，但是 N 位前面必须用数据填满。

Vector 的元素是用整型索引存取的，就像用指针一样，索引也是整型值，并且是从 0 开始的。

为了从 Vector 中重新得到数据索引也是从 0 到 size（）- 1 的。

为了在 Vector 设置数据索引也是从 0 到 size（）- 1 的。

为了在 Vector 插入数据索引也是从 0 到 size（）- 1 的。

1. 增加 Vector 的基础

在 Vector 的结尾增加一个 Vector：

```
addElement ( Object ) ;
```

下面是一个样例程序。为了使用 Vector 必须引入 Java：

```
import java.util. * ;

class VectorEg
{

  public static void main ( String [] args)
  {
    Vector names = new Vector ( 20, 5 );
    System . out . println ("capacity: "
        + names . capacity () );
    System . out . println ("size: "
        + names . size () );
    names . addElement ("Amy");
    names . addElement ("Bob");
    names . addElement ("Cindy");

    System . out . println ("capacity:"
        + names . capacity () );
    System . out . println ("size:"
        + names . size () );
  }
}
```

Vector 的初设容量是 20 和一个 5 的增量，并增加了 3 个涉及到的 String。就像程序给出的，增加的对象不是 Vector 的一部分。Vector 包括增加的对象的指针。

2. 删除元素

有时需要频繁地在表中删除元素，Vector 类中有一种方法可以删除元素而不在原处留下漏洞。

```
removeElementAt (int index)
```

元素在索引中的位置将被排除。元素子索引中的位置，索引 + 1，索引 + 2，……